Deitel® Series Page

How To Program Series
C How to Program, 6/E
C++ How to Program, 7/E
Java How to Program, 8/E, Early Objects Version
Java How to Program, 8/E, Late Objects Version
Internet & World Wide Web How to Program, 4/E
Visual Basic® 2008 How to Program
Visual C#® 2008 How to Program, 3/E
Visual C++® 2008 How to Program, 2/E
Small Java™ How to Program, 6/E
Small C++ How to Program, 5/E

Simply Series
Simply C++: An Application-Driven Tutorial Approach
Simply Java™ Programming: An Application-Driven Tutorial Approach
Simply C#: An Application-Driven Tutorial Approach

CourseSmart Web Books
www.deitel.com/books/CourseSmart.html
C++ How to Program, 5/E, 6/E & 7/E
Java How to Program, 6/E, 7/E & 8/E
Simply C++: An Application-Driven Tutorial Approach
Small C++ How to Program, 5/E
Small Java How to Program, 6/E
Visual Basic® 2008 How to Program
Visual C#® 2008 How to Program, 3/E
Deitel® Developer Series

AJAX, Rich Internet Applications and Web Development for Programmers
C++ for Programmers
C# 2008 for Programmers, 3/E

iPhone for Programmers: An App-Driven Approach
Java for Programmers
Javascript for Programmers

LiveLessons Video Learning Products

www.deitel.com/books/LiveLessons/

Java Fundamentals Parts 1 and 2
C# Fundamentals Parts 1 and 2

C++ Fundamentals Parts 1 and 2
JavaScript Fundamentals Parts 1 and 2

To receive updates on Deitel publications, Resource Centers, training courses, partner offers and more, please register for the free Deitel® Buzz Online e-mail newsletter at:

www.deitel.com/newsletter/subscribe.html

follow us on Twitter®
@deitel
and Facebook®
www.deitel.com/deitelfan/

To communicate with the authors, send e-mail to:
deitel@deitel.com

For information on government and corporate Dive-Into® Series on-site seminars offered by Deitel & Associates, Inc. worldwide, visit:

www.deitel.com/training/

or write to
deitel@deitel.com

For continuing updates on Prentice Hall/Deitel publications visit:

www.deitel.com
www.pearsonhighered.com/deitel

Check out our Resource Centers for valuable web resources that will help you master Java, other important programming languages, software and Internet- and web-related topics:

www.deitel.com/ResourceCenters.html
Trademarks

DEITEL, the double-thumbs-up bug and DIVE INTO are registered trademarks of Deitel and Associates, Inc.

Microsoft, Visual C++, Internet Explorer and the Windows logo are either registered trademarks or trade- marks of Microsoft Corporation in the United States and/or other countries.
In Memory of Clifford Stephens:

Your friendship, bright smile and infectious laugh will be truly missed.

Paul and Harvey Deitel
Deitel Resource Centers

Our Resource Centers focus on the vast amounts of free content available online. Find resources, downloads, tutorials, documentation, books, e-books, journals, articles, blogs, RSS feeds and more on many of today’s hottest programming and technology topics. For the most up-to-date list of our Resource Centers, visit:

www.deitel.com/ResourceCenters.html

Let us know what other Resource Centers you’d like to see! Also, please register for the free Deitel® Buzz Online e-mail newsletter at:

www.deitel.com/newsletter/subscribe.html

Computer Science
Functional Programming
Regular Expressions

Programming
ASP.NET 3.5
Adobe Flex
Ajax
Apex
ASP.NET Ajax
ASP.NET
C
C#
C++
C++ Boost Libraries
C++ Game Programming
C#
Code Search Engines and Code Sites
Computer Game Programming
CSS 2.1
Dojo
Facebook Developer Platform
Flash 9
Functional Programming
Java
Java Certification and Assessment Testing
Java Design Patterns
Java EE 5
Java SE 6
Java SE 7 (Dolphin)
Resource Center
JavaFX
JavaScript
JSON
Microsoft LINQ
Microsoft Popfly
.NET
.NET 3.0
.NET 3.5
Open Source
Perl
PHP
Programming Projects
Python
Regular Expressions
Ruby
Ruby on Rails
Silverlight

Games and Game Programming
Computer Game Programming
Computer Games
Mobile Gaming
Sudoku

Internet Business
Affiliate Programs
Competitive Analysis
Facebook Social Ads
Google AdSense
Google Analytics
Google Services
Internet Advertising
Internet Business Initiative
Internet Public Relations
Link Building
Location-Based Services
Online Lead Generation
Podcasting
Search Engine Optimization
Selling Digital Content
Sitemaps
Web Analytics
Website Monetization
YouTube and AdSense

Java
Java
Java Certification and Assessment Testing
Java Design Patterns
Java EE 5
Java SE 6
Java SE 7 (Dolphin)
Resource Center
JavaFX

Microsoft
ASP.NET
ASP.NET 3.5
ASP.NET Ajax
C#
DotNetNuke (DNN)
Internet Explorer 7 (IE7)
Microsoft LINQ
.NET
.NET 3.0
.NET 3.5
SharePoint
Silverlight
Visual Basic
Visual C++
Visual Studio Team System
Windows Presentation Foundation
Windows Vista

Apple
iPhone
Objective-C
Cocoa

Open Source & LAMP Stack
Apache
DotNetNuke (DNN)
Eclipse
Firefox
Linux
MySQL
Open Source Perl
PHP
Python
Ruby

Software
Apache
DotNetNuke (DNN)
Eclipse
Firefox
Internet Explorer 7 (IE7)
Linux
MySQL

Dive Into® Web 2.0 eBook
Web 2 eBook

Other Topics
Computer Games
Computing Jobs
Gadgets and Gizmos
Ring Tones
Sudoku
Appendices E through I are PDF documents posted online at the book’s Companion Website (located at www.pearsonhighered.com/deitel).

Preface

1 Introduction to Computers, the Internet and the Web

1.1 Introduction
1.2 Computers: Hardware and Software
1.3 Computer Organization
1.4 Personal, Distributed and Client/Server Computing
1.5 The Internet and the World Wide Web
1.6 Machine Languages, Assembly Languages and High-Level Languages
1.7 History of C
1.8 C Standard Library
1.9 C++
1.10 Java
1.11 Fortran, COBOL, Pascal and Ada
1.12 BASIC, Visual Basic, Visual C++, C# and .NET
1.13 Key Software Trend: Object Technology
1.14 Typical C Program Development Environment
1.15 Hardware Trends
1.16 Notes About C and This Book
1.17 Web Resources

2 Introduction to C Programming

2.1 Introduction
2.2 A Simple C Program: Printing a Line of Text
2.3 Another Simple C Program: Adding Two Integers
2.4 Memory Concepts
2.5 Arithmetic in C
2.6 Decision Making: Equality and Relational Operators

3 Structured Program Development in C

3.1 Introduction
3.2 Algorithms
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>195</td>
</tr>
<tr>
<td>6.2</td>
<td>Arrays</td>
<td>196</td>
</tr>
<tr>
<td>6.3</td>
<td>Defining Arrays</td>
<td>196</td>
</tr>
<tr>
<td>6.4</td>
<td>Array Examples</td>
<td>198</td>
</tr>
<tr>
<td>6.5</td>
<td>Passing Arrays to Functions</td>
<td>198</td>
</tr>
<tr>
<td>6.6</td>
<td>Sorting Arrays</td>
<td>212</td>
</tr>
<tr>
<td>6.7</td>
<td>Case Study: Computing Mean, Median and Mode Using Arrays</td>
<td>216</td>
</tr>
<tr>
<td>6.8</td>
<td>Searching Arrays</td>
<td>218</td>
</tr>
<tr>
<td>6.9</td>
<td>Multiple-Subscripted Arrays</td>
<td>223</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>253</td>
</tr>
<tr>
<td>7.2</td>
<td>Pointer Variable Definitions and Initialization</td>
<td>254</td>
</tr>
<tr>
<td>7.3</td>
<td>Pointer Operators</td>
<td>254</td>
</tr>
<tr>
<td>7.4</td>
<td>Passing Arguments to Functions by Reference</td>
<td>255</td>
</tr>
<tr>
<td>7.5</td>
<td>Using the const Qualifier with Pointers</td>
<td>257</td>
</tr>
<tr>
<td>7.6</td>
<td>Bubble Sort Using Call-by-Reference</td>
<td>261</td>
</tr>
<tr>
<td>7.7</td>
<td>sizeof Operator</td>
<td>267</td>
</tr>
<tr>
<td>7.8</td>
<td>Pointer Expressions and Pointer Arithmetic</td>
<td>270</td>
</tr>
<tr>
<td>7.9</td>
<td>Relationship between Pointers and Arrays</td>
<td>273</td>
</tr>
<tr>
<td>7.10</td>
<td>Arrays of Pointers</td>
<td>275</td>
</tr>
<tr>
<td>7.11</td>
<td>Case Study: Card Shuffling and Dealing Simulation</td>
<td>280</td>
</tr>
<tr>
<td>7.12</td>
<td>Pointers to Functions</td>
<td>285</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>309</td>
</tr>
<tr>
<td>8.2</td>
<td>Fundamentals of Strings and Characters</td>
<td>310</td>
</tr>
<tr>
<td>8.3</td>
<td>Character-Handling Library</td>
<td>310</td>
</tr>
<tr>
<td>8.4</td>
<td>String-Conversion Functions</td>
<td>312</td>
</tr>
<tr>
<td>8.5</td>
<td>Standard Input/Output Library Functions</td>
<td>317</td>
</tr>
<tr>
<td>8.6</td>
<td>String-Manipulation Functions of the String-Handling Library</td>
<td>322</td>
</tr>
<tr>
<td>8.7</td>
<td>Comparison Functions of the String-Handling Library</td>
<td>326</td>
</tr>
<tr>
<td>8.8</td>
<td>Search Functions of the String-Handling Library</td>
<td>329</td>
</tr>
<tr>
<td>8.9</td>
<td>Memory Functions of the String-Handling Library</td>
<td>331</td>
</tr>
<tr>
<td>8.10</td>
<td>Other Functions of the String-Handling Library</td>
<td>337</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>356</td>
</tr>
<tr>
<td>9.2</td>
<td>Streams</td>
<td>357</td>
</tr>
<tr>
<td>9.3</td>
<td>Formatting Output with printf</td>
<td>357</td>
</tr>
<tr>
<td>9.4</td>
<td>Printing Integers</td>
<td>358</td>
</tr>
<tr>
<td>9.5</td>
<td>Printing Floating-Point Numbers</td>
<td>359</td>
</tr>
</tbody>
</table>
9.6 Printing Strings and Characters  361
9.7 Other Conversion Specifiers  362
9.8 Printing with Field Widths and Precision  363
9.9 Using Flags in the printf Format Control String  366
9.10 Printing Literals and Escape Sequences  368
9.11 Reading Formatted Input with scanf  369

10 C Structures, Unions, Bit Manipulations and Enumerations  382
10.1 Introduction  383
10.2 Structure Definitions  383
10.3 Initializing Structures  386
10.4 Accessing Structure Members  386
10.5 Using Structures with Functions  388
10.6 typedef  388
10.7 Example: High-Performance Card Shuffling and Dealing Simulation  389
10.8 Unions  391
10.9 Bitwise Operators  394
10.10 Bit Fields  403
10.11 Enumeration Constants  406

11 C File Processing  417
11.1 Introduction  418
11.2 Data Hierarchy  418
11.3 Files and Streams  420
11.4 Creating a Sequential-Access File  421
11.5 Reading Data from a Sequential-Access File  426
11.6 Random-Access Files  430
11.7 Creating a Random-Access File  431
11.8 Writing Data Randomly to a Random-Access File  433
11.9 Reading Data from a Random-Access File  436
11.10 Case Study: Transaction-Processing Program  437

12 C Data Structures  454
12.1 Introduction  455
12.2 Self-Referential Structures  456
12.3 Dynamic Memory Allocation  456
12.4 Linked Lists  458
12.5 Stacks  466
12.6 Queues  472
12.7 Trees  478

13 C Preprocessor  495
13.1 Introduction  496
13.2 #include Preprocessor Directive  496
13.3 #define Preprocessor Directive: Symbolic Constants  496
13.4 #define Preprocessor Directive: Macros  497
13.5 Conditional Compilation  499
13.6 #error and #pragma Preprocessor Directives  500
13.7 # and ## Operators  500
13.8 Line Numbers  501
13.9 Predefined Symbolic Constants  501
13.10 Assertions  502

14 Other C Topics  507
14.1 Introduction  508
14.2 Redirecting I/O  508
14.3 Variable-Length Argument Lists  509
14.4 Using Command-Line Arguments  511
14.5 Notes on Compiling Multiple-Source-File Programs  512
14.6 Program Termination with exit and atexit  514
14.7 volatile Type Qualifier  515
14.8 Suffixes for Integer and Floating-Point Constants  516
14.9 More on Files  516
14.10 Signal Handling  518
14.11 Dynamic Memory Allocation: Functions calloc and realloc  520
14.12 Unconditional Branching with goto  521

15 C++ as a Better C; Introducing Object Technology  528
15.1 Introduction  529
15.2 C++  529
15.3 A Simple Program: Adding Two Integers  530
15.4 C++ Standard Library  532
15.5 Header Files  533
15.6 Inline Functions  535
15.7 References and Reference Parameters  537
15.8 Empty Parameter Lists  542
15.9 Default Arguments  542
15.10 Unary Scope Resolution Operator  544
15.11 Function Overloading  545
15.12 Function Templates  548
15.13 Introduction to Object Technology and the UML  551
15.14 Wrap-Up  554

16 Introduction to Classes and Objects  560
16.1 Introduction  561
16.2 Classes, Objects, Member Functions and Data Members  561
16.3 Defining a Class with a Member Function  562
16.4 Defining a Member Function with a Parameter  566
16.5 Data Members, set Functions and get Functions  569
16.6 Initializing Objects with Constructors  576
16.7 Placing a Class in a Separate File for Reusability  579
16.8 Separating Interface from Implementation  583
16.9 Validating Data with set Functions  589
16.10 Wrap-Up  594

17 **Classes: A Deeper Look, Part 1**  601
17.1 Introduction  602
17.2 Time Class Case Study  603
17.3 Class Scope and Accessing Class Members  609
17.4 Separating Interface from Implementation  611
17.5 Access Functions and Utility Functions  612
17.6 Time Class Case Study: Constructors with Default Arguments  615
17.7 Destructors  620
17.8 When Constructors and Destructors are Called  621
17.9 Time Class Case Study: A Subtle Trap—Returning a Reference to a private Data Member  624
17.10 Default Memberwise Assignment  627
17.11 Wrap-Up  629

18 **Classes: A Deeper Look, Part 2**  635
18.1 Introduction  636
18.2 const (Constant) Objects and const Member Functions  636
18.3 Composition: Objects as Members of Classes  645
18.4 friend Functions and friend Classes  651
18.5 Using the this Pointer  654
18.6 static Class Members  659
18.7 Data Abstraction and Information Hiding  664
18.8 Wrap-Up  666

19 **Operator Overloading**  672
19.1 Introduction  673
19.2 Fundamentals of Operator Overloading  674
19.3 Restrictions on Operator Overloading  675
19.4 Operator Functions as Class Members vs. Global Function  676
19.5 Overloading Stream Insertion and Stream Extraction Operators  678
19.6 Overloading Unary Operators  681
19.7 Overloading Binary Operators  682
19.8 Dynamic Memory Management  682
19.9 Case Study: Array Class  684
19.10 Converting between Types  696

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.3</td>
<td>Defining a Class with a Member Function</td>
<td>562</td>
</tr>
<tr>
<td>16.4</td>
<td>Defining a Member Function with a Parameter</td>
<td>566</td>
</tr>
<tr>
<td>16.5</td>
<td>Data Members, set Functions and get Functions</td>
<td>569</td>
</tr>
<tr>
<td>16.6</td>
<td>Initializing Objects with Constructors</td>
<td>576</td>
</tr>
<tr>
<td>16.7</td>
<td>Placing a Class in a Separate File for Reusability</td>
<td>579</td>
</tr>
<tr>
<td>16.8</td>
<td>Separating Interface from Implementation</td>
<td>583</td>
</tr>
<tr>
<td>16.9</td>
<td>Validating Data with set Functions</td>
<td>589</td>
</tr>
<tr>
<td>16.10</td>
<td>Wrap-Up</td>
<td>594</td>
</tr>
<tr>
<td>17</td>
<td><strong>Classes: A Deeper Look, Part 1</strong></td>
<td>601</td>
</tr>
<tr>
<td>17.1</td>
<td>Introduction</td>
<td>602</td>
</tr>
<tr>
<td>17.2</td>
<td>Time Class Case Study</td>
<td>603</td>
</tr>
<tr>
<td>17.3</td>
<td>Class Scope and Accessing Class Members</td>
<td>609</td>
</tr>
<tr>
<td>17.4</td>
<td>Separating Interface from Implementation</td>
<td>611</td>
</tr>
<tr>
<td>17.5</td>
<td>Access Functions and Utility Functions</td>
<td>612</td>
</tr>
<tr>
<td>17.6</td>
<td>Time Class Case Study: Constructors with Default Arguments</td>
<td>615</td>
</tr>
<tr>
<td>17.7</td>
<td>Destructors</td>
<td>620</td>
</tr>
<tr>
<td>17.8</td>
<td>When Constructors and Destructors are Called</td>
<td>621</td>
</tr>
<tr>
<td>17.9</td>
<td>Time Class Case Study: A Subtle Trap—Returning a Reference to a private Data Member</td>
<td>624</td>
</tr>
<tr>
<td>17.10</td>
<td>Default Memberwise Assignment</td>
<td>627</td>
</tr>
<tr>
<td>17.11</td>
<td>Wrap-Up</td>
<td>629</td>
</tr>
<tr>
<td>18</td>
<td><strong>Classes: A Deeper Look, Part 2</strong></td>
<td>635</td>
</tr>
<tr>
<td>18.1</td>
<td>Introduction</td>
<td>636</td>
</tr>
<tr>
<td>18.2</td>
<td>const (Constant) Objects and const Member Functions</td>
<td>636</td>
</tr>
<tr>
<td>18.3</td>
<td>Composition: Objects as Members of Classes</td>
<td>645</td>
</tr>
<tr>
<td>18.4</td>
<td>friend Functions and friend Classes</td>
<td>651</td>
</tr>
<tr>
<td>18.5</td>
<td>Using the this Pointer</td>
<td>654</td>
</tr>
<tr>
<td>18.6</td>
<td>static Class Members</td>
<td>659</td>
</tr>
<tr>
<td>18.7</td>
<td>Data Abstraction and Information Hiding</td>
<td>664</td>
</tr>
<tr>
<td>18.8</td>
<td>Wrap-Up</td>
<td>666</td>
</tr>
<tr>
<td>19</td>
<td><strong>Operator Overloading</strong></td>
<td>672</td>
</tr>
<tr>
<td>19.1</td>
<td>Introduction</td>
<td>673</td>
</tr>
<tr>
<td>19.2</td>
<td>Fundamentals of Operator Overloading</td>
<td>674</td>
</tr>
<tr>
<td>19.3</td>
<td>Restrictions on Operator Overloading</td>
<td>675</td>
</tr>
<tr>
<td>19.4</td>
<td>Operator Functions as Class Members vs. Global Function</td>
<td>676</td>
</tr>
<tr>
<td>19.5</td>
<td>Overloading Stream Insertion and Stream Extraction Operators</td>
<td>678</td>
</tr>
<tr>
<td>19.6</td>
<td>Overloading Unary Operators</td>
<td>681</td>
</tr>
<tr>
<td>19.7</td>
<td>Overloading Binary Operators</td>
<td>682</td>
</tr>
<tr>
<td>19.8</td>
<td>Dynamic Memory Management</td>
<td>682</td>
</tr>
<tr>
<td>19.9</td>
<td>Case Study: Array Class</td>
<td>684</td>
</tr>
<tr>
<td>19.10</td>
<td>Converting between Types</td>
<td>696</td>
</tr>
</tbody>
</table>
## 19.11 Building a String Class 697
19.12 Overloading ++ and -- 698
19.13 Case Study: A Date Class 700
19.14 Standard Library Class string 704
19.15 explicit Constructors 708
19.16 Proxy Classes 711
19.17 Wrap-Up 715

### 20 Object-Oriented Programming: Inheritance 727
20.1 Introduction 728
20.2 Base Classes and Derived Classes 729
20.3 protected Members 732
20.4 Relationship between Base Classes and Derived Classes 732
   20.4.1 Creating and Using a CommissionEmployee Class 733
   20.4.2 Creating a BasePlusCommissionEmployee Class Without Using Inheritance 738
   20.4.3 Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy 743
   20.4.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Data 748
   20.4.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Data 755
20.5 Constructors and Destructors in Derived Classes 762
20.6 public, protected and private Inheritance 770
20.7 Software Engineering with Inheritance 771
20.8 Wrap-Up 772

### 21 Object-Oriented Programming: Polymorphism 778
21.1 Introduction 779
21.2 Polymorphism Examples 780
21.3 Relationships Among Objects in an Inheritance Hierarchy 781
   21.3.1 Invoking Base-Class Functions from Derived-Class Objects 782
   21.3.2 Aiming Derived-Class Pointers at Base-Class Objects 789
   21.3.3 Derived-Class Member-Function Calls via Base-Class Pointers 790
   21.3.4 Virtual Functions 792
   21.3.5 Summary of the Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers 798
21.4 Type Fields and switch Statements 799
21.5 Abstract Classes and Pure virtual Functions 799
21.6 Case Study: Payroll System Using Polymorphism 801
   21.6.1 Creating Abstract Base Class Employee 803
   21.6.2 Creating Concrete Derived Class SalariedEmployee 806
   21.6.3 Creating Concrete Derived Class HourlyEmployee 808
   21.6.4 Creating Concrete Derived Class CommissionEmployee 811
### Contents

- **D.4** Programming Sudoku Puzzle Solvers 942
- **D.5** Generating New Sudoku Puzzles 943
- **D.6** Conclusion 945

### Appendices on the Web 946

Appendices E through I are PDF documents posted online at the book’s Companion Website (located at [www.pearsonhighered.com/deitel](http://www.pearsonhighered.com/deitel)).

#### E Game Programming with the Allegro C Library I

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.1 Introduction</td>
<td>II</td>
</tr>
<tr>
<td>E.2 Installing Allegro</td>
<td>II</td>
</tr>
<tr>
<td>E.3 A Simple Allegro Program</td>
<td>III</td>
</tr>
<tr>
<td>E.4 Simple Graphics: Importing Bitmaps and Blitting</td>
<td>IV</td>
</tr>
<tr>
<td>E.5 Animation with Double Buffering</td>
<td>IX</td>
</tr>
<tr>
<td>E.6 Importing and Playing Sounds</td>
<td>XVI</td>
</tr>
<tr>
<td>E.7 Keyboard Input</td>
<td>XX</td>
</tr>
<tr>
<td>E.8 Fonts and Displaying Text</td>
<td>XXV</td>
</tr>
<tr>
<td>E.9 Implementing the Game of Pong</td>
<td>XXXI</td>
</tr>
<tr>
<td>E.10 Timers in Allegro</td>
<td>XXXVII</td>
</tr>
<tr>
<td>E.11 The Grabber and Allegro Datafiles</td>
<td>XLII</td>
</tr>
<tr>
<td>E.12 Other Allegro Capabilities</td>
<td>LI</td>
</tr>
<tr>
<td>E.13 Allegro Resource Center</td>
<td>LII</td>
</tr>
</tbody>
</table>

#### F Sorting: A Deeper Look LVIII

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>F.1 Introduction</td>
<td>LIX</td>
</tr>
<tr>
<td>F.2 Big O Notation</td>
<td>LIX</td>
</tr>
<tr>
<td>F.3 Selection Sort</td>
<td>LX</td>
</tr>
<tr>
<td>F.4 Insertion Sort</td>
<td>LXIV</td>
</tr>
<tr>
<td>F.5 Merge Sort</td>
<td>LXVII</td>
</tr>
</tbody>
</table>

#### G Introduction to C99 LXXVIII

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>G.1 Introduction</td>
<td>LXXIX</td>
</tr>
<tr>
<td>G.2 Support for C99</td>
<td>LXXX</td>
</tr>
<tr>
<td>G.3 New C99 Headers</td>
<td>LXXX</td>
</tr>
<tr>
<td>G.4 // Comments</td>
<td>LXXX</td>
</tr>
<tr>
<td>G.5 Mixing Declarations and Executable Code</td>
<td>LXXXI</td>
</tr>
<tr>
<td>G.6 Declaring a Variable in a for Statement Header</td>
<td>LXXXII</td>
</tr>
<tr>
<td>G.7 Designated Initializers and Compound Literals</td>
<td>LXXXIV</td>
</tr>
<tr>
<td>G.8 Type bool</td>
<td>LXXXVII</td>
</tr>
<tr>
<td>G.9 Implicit int in Function Declarations</td>
<td>LXXXVIII</td>
</tr>
<tr>
<td>G.10 Complex Numbers</td>
<td>LXXXIX</td>
</tr>
<tr>
<td>G.11 Variable-Length Arrays</td>
<td>XC</td>
</tr>
</tbody>
</table>
Welcome to the C programming language—and to C++, too! This book presents leading-edge computing technologies for students, instructors and software development professionals.

At the heart of the book is the Deitel signature “live-code approach.” Concepts are presented in the context of complete working programs, rather than in code snippets. Each code example is immediately followed by one or more sample executions. All the source code is available at www.deitel.com/books/cht6/.

We believe that this book and its support materials will give you an informative, interesting, challenging and entertaining introduction to C.

As you read the book, if you have questions, send an e-mail to deitel@deitel.com; we’ll respond promptly. For updates on this book and its supporting C and C++ software, and for the latest news on all Deitel publications and services, visit www.deitel.com.

New and Updated Features

Here are the updates we’ve made for C How to Program, 6/e:

- **“Making a Difference” Exercises Set.** We encourage you to use computers and the Internet to research and solve problems that really matter. These new exercises are meant to increase awareness of important issues the world is facing. We hope you’ll approach them with your own values, politics and beliefs.

- **Tested All Code on Windows and Linux.** We’ve tested every program (the examples and the exercises) using both Visual C++ 2008 and GNU GCC 4.3. The code examples and exercise code solutions were also tested using Visual Studio 2010 Beta.

- **New Design.** The book has a new interior design that graphically serves to organize, clarify and highlight the information, and enhances the book’s pedagogy.

- **Improved Terminology Sections.** We’ve added page numbers for the defining occurrences of all terms in the terminology lists for easy reference.

- **Updated Coverage of C++ and Object-Oriented Programming.** We updated Chapters 15–24 on object-oriented programming in C++ with material from our just published C++ How to Program, 7/e.

- **Titled Programming Exercises.** We’ve titled all the programming exercises. This helps instructors tune assignments for their classes.

- **New Web Appendices.** Chapters 15–17 from the previous edition are now searchable PDF Appendices E–G, available on the Companion Website (see the access card at the front of the book).
New Debugger Appendices. We also added new debugging appendices for Visual C++® 2008 and GNU gdb.

Order of Evaluation. We added cautions about order of evaluation issues.

We replaced all uses of gets (from <stdio.h>) with fgets, because gets is now deprecated.

Additional Exercises. We added more function pointer exercises. We also added the Fibonacci exercise project that improves the Fibonacci recursion example (tail recursion).

Secure C Programming Resource Center. We’ve posted a new Secure C Programming Resource Center at www.deitel.com/SecureC/. We’ve also added notes about secure C programming to the introductions in Chapter 7, Pointers, and Chapter 8, Strings.

Game Programming with Allegro. We updated the chapter on game programming with the Allegro C library. In particular, we added instructions on installing the Allegro libraries for use with Visual C++® 2008 and GNU GCC 4.3.

Coverage of the C99 Standard. We updated and enhanced the detailed appendix on C99, which was reviewed by John Benito, Convener of ISO WG14—the Working Group responsible for the C Programming Language Standard. Each C99 concept is now keyed to the section where it can be taught earlier in the book. C99 is not incorporated throughout the book because Microsoft does not yet support it and a large percentage of C courses use Microsoft’s Visual C++® compiler. For additional information, check out the C99 Standard section in our C Resource center at www.deitel.com/C/. You’ll find features of C99, articles from experts, the differences between Standard C and C99, FAQs, downloads and more.

C++-Style // Comments. We discuss C++-style // comments early for instructors and students who’d prefer to use them. Although Microsoft C does not yet support C99, it does support C99’s comments, which are borrowed from C++.

C Standard Library. Section 1.8 now references P.J. Plauger’s Dinkumware website (www.dinkumware.com/manuals/default.aspx) where students can find thorough searchable documentation for the C Standard Library functions.

Other Features

Other features of C How to Program, 6/e, include:

Game Programming with the Allegro C Game Programming Library

Appendix E introduces the Allegro game programming C library. This library—originally developed by Climax game programmer Shawn Hargreaves—was created to be a powerful tool for programming games in C while still remaining relatively simple compared to other, more complicated graphics libraries such as DirectX and OpenGL. In Appendix E, we use Allegro’s capabilities to create the simple game of Pong. Along the way, we demonstrate how to display graphics, play sounds, receive input from the keyboard and create timed events—features you can use to create games of your own. Students and instructors
alike will find Allegro challenging and entertaining. We include extensive web resources in our Allegro Resource Center (www.deitel.com/allegro), one of which offers more than 1000 open-source Allegro games.

**Sorting: A Deeper Look**
Sorting places data in order, based on one or more sort keys. We begin our presentation of sorting with a simple algorithm in Chapter 6. In Appendix F, we present a deeper look at sorting. We consider several algorithms and compare them with regard to their memory consumption and processor demands. For this purpose, we introduce Big O notation, which indicates how hard an algorithm may have to work to solve a problem. Through examples and exercises, Appendix F discusses the selection sort, insertion sort, recursive merge sort, recursive selection sort, bucket sort and recursive Quicksort.

**Web-Based Materials**
This book is supported by substantial online materials. The book’s Companion Website (www.pearsonhighered.com/deitel; see the access card at the front of the book) contains the following appendices in searchable PDF format:

- Appendix E, Game Programming with the Allegro C Library
- Appendix F, Sorting: A Deeper Look
- Appendix G, Introduction to C99
- Appendix H, Using the Visual Studio Debugger
- Appendix I, Using the GNU Debugger

**Dependency Charts**
The dependency charts in Figs. 1–2 show the dependencies among the chapters to help instructors plan their syllabi. *C How to Program, 6/e* is appropriate for CS1 and CS2 courses, and intermediate-level C and C++ programming courses. The C++ part of the book assumes that you have studied the C part.

**Teaching Approach**
*C How to Program, 6/e*, contains a rich collection of examples. We concentrate on demonstrating the principles of good software engineering and stressing program clarity.

**Live-Code Approach.** *C How to Program, 6/e*, is loaded with “live-code” examples. Most new concepts are presented in the context of complete working C applications, followed by one or more executions showing program inputs and outputs.

**Syntax Shading**
For readability, we syntax shade the code, similar to the way most integrated-development environments and code editors syntax color code. Our syntax-shading conventions are:

- comments appear like this
- keywords appear like this
- constants and literal values appear like this
- all other code appears in black
Preface

C Chapter Dependency Chart

[Note: Arrows pointing into a chapter indicate that chapter's dependencies.]

Fig. 1 | C chapter dependency chart.

Code Highlighting
We place gray rectangles around the key code.

Using Fonts for Emphasis. We place the key terms and the index’s page reference for each defining occurrence in bold blue text for easy reference. We emphasize on-screen components in the bold Helvetica font (e.g., the File menu) and C program text in the Lucida font (for example, int x = 5;).

Web Access. All of the source-code examples are available for download from:

www.deitel.com/books/chtp6/
Teaching Approach

**Quotations.** Each chapter begins with quotations. We hope that you enjoy relating these to the chapter material.

**Objectives.** The quotes are followed by a list of chapter objectives.

**Illustrations/Figures.** Abundant charts, tables, line drawings, UML diagrams, programs and program output are included.

**Programming Tips**

We include programming tips to help you focus on important aspects of program development. These tips and practices represent the best we’ve gleaned from a combined seven decades of programming and teaching experience.

---

**Good Programming Practice**

*The Good Programming Practices call attention to techniques that will help you produce programs that are clearer, more understandable and more maintainable.*

**Common Programming Error**

*Pointing out these Common Programming Errors reduces the likelihood that you’ll make them.*

**Error-Prevention Tip**

*These tips contain suggestions for exposing and removing bugs from your programs; many describe aspects of C that prevent bugs from getting into programs in the first place.*
Summary Bullets. We present a section-by-section, bullet-list summary of the chapter.

Terminology. We include an alphabetized list of the important terms defined in each chapter with the page number of each term’s defining occurrence for easy reference.

Self-Review Exercises and Answers. Extensive self-review exercises and answers are included for self-study.

Exercises. Each chapter concludes with a substantial set of exercises including:

- simple recall of important terminology and concepts,
- identifying the errors in code samples,
- writing individual C statements,
- writing small portions of functions and classes,
- writing complete C functions, classes and programs, and
- major projects.

Instructors can use these exercises to form homework assignments, short quizzes, major examinations and term projects. [NOTE: Please do not write to us requesting access to the Pearson Instructor’s Resource Center which contains the book’s instructor supplements, including the exercise solutions. Access is limited strictly to college instructors teaching from the book. Instructors may obtain access only through their Pearson representatives. Solutions are not provided for “project” exercises.] Check out our Programming Projects Resource Center (www.deitel.com/ProgrammingProjects/) for lots of additional exercise and project possibilities.

Index. We’ve included an extensive index, which is especially useful when you use the book as a reference. Defining occurrences of key terms are highlighted with a bold blue page number.

Student Resources

Many C and C++ development tools are available. We wrote C How to Program, 6/e primarily using Microsoft’s free Visual C++® Express Edition (which is available free for download at www.microsoft.com/express/vc/) and the free GNU C++ (gcc.gnu.org/install/binaries.html), which is already installed on most Linux systems and can be installed on
Mac OS X and Windows systems as well. You can learn more about Visual C++® Express at msdn.microsoft.com/vstudio/express/visualc. You can learn more about GNU C++ at gcc.gnu.org. Apple includes GNU C++ in their Xcode development tools, which Mac OS X users can download from developer.apple.com/tools/xcode.

You can download the book’s examples and additional resources from:

www.deitel.com/books/chtp6/

For additional resources and software downloads see our C Resource Center:

www.deitel.com/c/

For other C and C++ compilers that are available free for download:

www.thefreecountry.com/developercity/ccompilers.shtml
www.compilers.net/Dir/Compilers/CCpp.htm

CourseSmart Web Books

Today’s students and instructors have increasing demands on their time and money. Pearson has responded to that need by offering digital texts and course materials online through CourseSmart. CourseSmart allows faculty to review course materials online saving time and costs. It is also environmentally sound and offers students a high-quality digital version of the text for as much as 50% off the cost of a print copy of the text. Students receive the same content offered in the print textbook enhanced by search, note-taking, and printing tools. For more information, visit www.coursesmart.com.

Software for the Book

This book includes the Microsoft® Visual Studio® 2008 Express Editions All-in-One DVD, which contains the Visual C++® 2008 Express Edition (and other Microsoft development tools). You can also download the latest version of Visual C++ Express Edition from:

www.microsoft.com/express/vc

Per Microsoft’s website, Express Editions are “lightweight, easy-to-use and easy-to-learn tools for the hobbyist, novice and student developer.” They are appropriate for academic courses and for professionals who do not have access to a complete version of Visual Studio 2008.

With the exception of one example in Chapter 9, C Formatted Input/Output, and the examples in Appendix G, Introduction to C99, all of the examples in this book compile and run in Visual C++® 2008 and the beta version of Visual C++® 2010. All of the examples compile and run in GNU GCC 4.3. GCC is available for most platforms, including Linux, Mac OS X (via Xcode) and Windows—via tools like Cygwin (www.cygwin.com) and MinGW (www.mingw.org).

Instructor Supplements

The following supplements are available to qualified instructors only through Pearson Education’s Instructor Resource Center (www.pearsonhighered.com/irc):

• Solutions Manual with solutions to most of the end-of-chapter exercises.
Preface

- Test Item File of multiple-choice questions (approximately two per book section)
- Customizable PowerPoint® slides containing all the code and figures in the text, plus bulleted items that summarize key points in the text

If you are not already a registered faculty member, contact your Pearson representative or visit www.pearsonhighered.com/educator/replocator/.

Deitel® Buzz Online Free E-mail Newsletter

The Deitel® Buzz Online e-mail newsletter will keep you posted about issues related to C How to Program, 6/e. It also includes commentary on industry trends and developments, links to free articles and resources from our published books and upcoming publications, product-release schedules, errata, challenges, anecdotes, information on our corporate instructor-led training courses and more. To subscribe, visit www.deitel.com/newsletter/subscribe.html

The Deitel Online Resource Centers

Our website www.deitel.com provides more than 100 Resource Centers on various topics including programming languages, software development, Web 2.0, Internet business and open-source projects—see the list of Resource Centers in the first few pages of this book and visit www.deitel.com/ResourceCenters.html. We’ve found many exceptional resources online, including tutorials, documentation, software downloads, articles, blogs, podcasts, videos, code samples, books, e-books and more—most of them are free. Each week we announce our latest Resource Centers in our newsletter, the Deitel® Buzz Online. Some of the Resource Centers you might find helpful while studying this book are C, C++, C++ Boost Libraries, C++ Game Programming, Visual C++, UML, Code Search Engines and Code Sites, Game Programming and Programming Projects.

Follow Deitel on Twitter and Facebook

To receive updates on Deitel publications, Resource Centers, training courses, partner offers and more, follow us on Twitter®

@deitel

and join the Deitel & Associates group on Facebook®

www.deitel.com/deitelfan/

Acknowledgments

It’s a pleasure to acknowledge the efforts of people whose names do not appear on the cover, but whose hard work, cooperation, friendship and understanding were crucial to the book’s production. Many people at Deitel & Associates, Inc., devoted long hours to this project—thanks especially to Abbey Deitel and Barbara Deitel.

We would also like to thank the participants of our Honors Internship program who contributed to this publication—Christine Chen, an Operations Research and Information Engineering major at Cornell University; and Matthew Pearson, a Computer Science graduate of Cornell University.
We are fortunate to have worked on this project with the dedicated team of publishing professionals at Pearson. We appreciate the efforts of Marcia Horton, Editorial Director of Pearson’s Engineering and Computer Science Division, and Michael Hirsch, Editor-in-Chief of Computer Science. Carole Snyder recruited the book’s review team and managed the review process. Francesco Santalucía (an independent artist) and Kristine Carney of Pearson designed the book’s cover—we provided the concept, and they made it happen. Scott Disanno and Bob Engelhardt managed the book’s production. Erin Davis and Margaret Waples marketed the book through academic and professional channels.

C How to Program, 6/e Reviewers
We wish to acknowledge the efforts of our reviewers. Adhering to a tight time schedule, they scrutinized the text and the programs and provided countless suggestions for improving the accuracy and completeness of the presentation:

- Xiaolong Li, Indiana State University
- Tom Rethard, The University of Texas at Arlington

C How to Program, 5/e Reviewers
- Alireza Fazelpour (Palm Beach Community College)
- Don Kostuch (Independent Consultant)
- Ed James Beckham (Altera)
- Gary Sibbitts (St. Louis Community College at Meramec)
- Ian Barland (Radford University)
- Kevin Mark Jones (Hewlett Packard)
- Mahesh Hariharan (Microsoft)
- William Mike Miller (Edison Design Group, Inc.)
- Benjamin Seyfarth (University of Southern Mississippi)
- William Albrecht (University of South Florida)
- William Smith (Tulsa Community College)

Allegro Reviewers for C How to Program, 5/e
- Shawn Hargreaves (Software Design Engineer, Microsoft Xbox)
- Matthew Leverton (Founder and Webmaster of Allegro.cc)
- Ryan Patterson (Independent Consultant)
- Douglas Walls (Senior Staff Engineer, C compiler, Sun Microsystems)

C99 Reviewers for C How to Program, 5/e
- Lawrence Jones, (UGS Corp.)
- Douglas Walls (Senior Staff Engineer, C compiler, Sun Microsystems)
Preface

Well, there you have it! C is a powerful programming language that will help you write programs quickly and effectively. C scales nicely into the realm of enterprise systems development to help organizations build their business-critical and mission-critical information systems. As you read the book, we would sincerely appreciate your comments, criticisms, corrections and suggestions for improving the text. Please address all correspondence to:

deitel@deitel.com

We'll respond promptly, and post corrections and clarifications on:

www.deitel.com/books/chtp6/

We hope you enjoy working with C How to Program, Sixth Edition as much as we enjoyed writing it!

Paul Deitel
Harvey Deitel
Maynard, Massachusetts
August 2009

About the Authors

Paul J. Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT’s Sloan School of Management, where he studied Information Technology. Through Deitel & Associates, Inc., he has delivered C, C++, Java, C#, Visual Basic and Internet programming courses to industry clients, including Cisco, IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Stratus, Cambridge Technology Partners, Open Environment Corporation, One Wave, Hyperion Software, Adra Systems, Entergy, Cable-Data Systems, Nortel Networks, Puma, iRobot, Invensys and many more. He holds the Java Certified Programmer and Java Certified Developer certifications and has been designated by Sun Microsystems as a Java Champion. He has also lectured on Java and C++ for the Boston Chapter of the Association for Computing Machinery. He and his co-author, Dr. Harvey M. Deitel, are the world’s best-selling programming-language textbook authors.

Dr. Harvey M. Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has 48 years of academic and industry experience in the computer field. Dr. Deitel earned B.S. and M.S. degrees from MIT and a Ph.D. from Boston University. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., with his son, Paul J. Deitel. He and Paul are the co-authors of dozens of books and multimedia packages and they are writing many more. With translations published in Japanese, German, Russian, Traditional Chinese, Simplified Chinese, Spanish, Korean, French, Polish, Italian, Portuguese, Greek, Urdu and Turkish, the Deitels’ texts have earned international recognition. Dr. Deitel has delivered hundreds of professional seminars to major corporations, academic institutions, government organizations and the military.
About Deitel & Associates, Inc.

Deitel & Associates, Inc., is an internationally recognized authoring and corporate training organization specializing in computer programming languages, Internet and web software technology, object-technology education and iPhone applications development. The company provides instructor-led courses delivered at client sites worldwide on major programming languages and platforms, such as C, C++, Visual C++®, Java™, Visual C#®, Visual Basic®, XML®, Python®, object technology, Internet and web programming, iPhone programming and a growing list of additional programming and software-development-related courses. The founders of Deitel & Associates, Inc., are Paul J. Deitel and Dr. Harvey M. Deitel. The company’s clients include many of the world’s largest companies, government agencies, branches of the military, and academic institutions. Through its 33-year publishing partnership with Prentice Hall/Pearson, Deitel & Associates, Inc., publishes leading-edge programming textbooks, professional books, interactive multimedia Cyber Classrooms, LiveLessons video courses (online at www.safaribooksonline.com and on DVD at www.deitel.com/books/livelessons/), and e-content for popular course-management systems.

Deitel & Associates, Inc., and the authors can be reached via e-mail at:

ditel@deitel.com

To learn more about Deitel & Associates, Inc., its publications and its Dive Into® Series Corporate Training curriculum delivered at client locations worldwide, visit:

www.deitel.com/training/

and subscribe to the free Deitel® Buzz Online e-mail newsletter at:

www.deitel.com/newsletter/subscribe.html

Individuals wishing to purchase Deitel books, and LiveLessons DVD and web-based training courses can do so through www.deitel.com. Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit www.prenhall.com/mischtm/support.html#order.
This page intentionally left blank
Introduction to Computers, the Internet and the Web

The chief merit of language is clearness.
—Galen

Our life is frittered away by detail. … Simplify, simplify.
—Henry David Thoreau

He had a wonderful talent for packing thought close, and rendering it portable.
—Thomas B. Macaulay

Man is still the most extraordinary computer of all.
—John F. Kennedy

Objectives
In this chapter, you’ll learn:

■ Basic computer concepts.
■ The different types of programming languages.
■ The history of the C programming language.
■ The purpose of the C Standard Library.
■ The elements of a typical C program development environment.
■ How C provides a foundation for further study of programming languages in general and of C++, Java and C# in particular.
■ The history of the Internet and the World Wide Web.
**1.1 Introduction**

Welcome to C and C++! We’ve worked hard to create what we hope you’ll find to be an informative, entertaining and challenging learning experience. C is a powerful computer programming language that is appropriate for technically oriented people with little or no programming experience and for experienced programmers to use in building substantial information systems. *C How to Program, Sixth Edition,* is an effective learning tool for each of these audiences.

The core of the book emphasizes achieving program clarity through the proven techniques of structured programming. You’ll learn programming the right way from the beginning. We’ve attempted to write in a clear and straightforward manner. The book is abundantly illustrated. Perhaps most important, the book presents hundreds of complete working programs and shows the outputs produced when those programs are run on a computer. We call this the “live-code approach.” All of these example programs may be downloaded from our website www.deitel.com/books/chtp6/.

Most people are familiar with the exciting tasks computers perform. Using this textbook, you’ll learn how to command computers to perform those tasks. It’s software (i.e., the instructions you write to command computers to perform actions and make decisions) that controls computers (often referred to as hardware). This text introduces programming in C, which was standardized in 1989 as ANSI X3.159-1989 in the United States through the American National Standards Institute (ANSI), then worldwide through the efforts of the International Standards Organization (ISO). We call this Standard C. We also introduce C99 (ISO/IEC 9899:1999)—the latest version of the C standard. C99 has not yet been universally adopted, so we chose to discuss it in (optional) Appendix G. A new C standard, which has been informally named C1X, is under development and likely to be published around 2012.

Optional Appendix E presents the Allegro game programming C library. The appendix shows how to use Allegro to create a simple game. We show how to display graphics and smoothly animate objects, and we explain additional features such as sound,
keyboard input and text output. The appendix includes web links and resources that point you to over 1000 Allegro games and to tutorials on advanced Allegro techniques.

Computer use is increasing in most fields of endeavor. Computing costs have decreased dramatically due to rapid developments in both hardware and software technologies. Computers that might have filled large rooms and cost millions of dollars a few decades ago can now be inscribed on silicon chips smaller than a fingernail, costing a few dollars each. Those large computers were called **mainframes** and current versions are widely used today in business, government and industry. Fortunately, silicon is one of the most abundant materials on earth—it’s an ingredient in common sand. Silicon chip technology has made computing so economical that more than a billion general-purpose computers are in use worldwide, helping people in business, industry and government, and in their personal lives. Billions more special purpose computers are used in intelligent electronic devices like car navigation systems, energy-saving appliances and game controllers.

C++, an object-oriented programming language based on C, is of such interest today that we’ve included a detailed introduction to C++ and object-oriented programming in Chapters 15–24. In the programming languages marketplace, many key vendors market a combined C/C++ product rather than offering separate products. This enables users to continue programming in C if they wish, then gradually migrate to C++ when it’s appropriate.

To keep up to date with C and C++ developments at Deitel & Associates, register for our free e-mail newsletter, the **Deitel® Buzz Online**, at

www.deitel.com/newsletter/subscribe.html

Check out our growing list of C and related Resource Centers at

www.deitel.com/ResourceCenters.html

Some Resource Centers that will be valuable to you as you read the C portion of this book are C, Code Search Engines and Code Sites, Computer Game Programming and Programming Projects. Each week we announce our latest Resource Centers in the newsletter. Errata and updates for this book are posted at

www.deitel.com/books/chtp6/

You’re embarking on a challenging and rewarding path. As you proceed, if you have any questions, send e-mail to

deitel@deitel.com

We’ll respond promptly. We hope that you’ll enjoy **C How to Program, Sixth Edition**.

### 1.2 Computers: Hardware and Software

A **computer** is a device that can perform computations and make logical decisions billions of times faster than human beings can. For example, many of today’s personal computers can perform several billion additions per second. A person operating a desk calculator could spend an entire lifetime performing calculations and still not complete as many calculations as a powerful personal computer can perform in one second! (Points to ponder: How would you know whether the person added the numbers correctly? How would you know whether the computer added the numbers correctly?) Today’s fastest **supercomputers** can perform **thousands of trillions (quadrillions)** of instructions per second! To put that
in perspective, a quadrillion-instruction-per-second computer can perform more than 100,000 calculations per second for every person on the planet!

Computers process data under the control of sets of instructions called computer programs. These programs guide the computer through orderly sets of actions specified by people called computer programmers.

A computer consists of various devices referred to as hardware (e.g., the keyboard, screen, mouse, hard disk, memory, DVDs and processing units). The programs that run on a computer are referred to as software. Hardware costs have been declining dramatically in recent years, to the point that personal computers have become a commodity. In this book, you’ll learn proven methods that are reducing software development costs—structured programming (in the C chapters) and object-oriented programming (in the C++ chapters).

**1.3 Computer Organization**

Regardless of differences in physical appearance, virtually every computer may be envisioned as divided into six logical units or sections:

1. **Input unit.** This “receiving” section obtains information (data and computer programs) from input devices and places it at the disposal of the other units so that it can be processed. Humans typically enter information into computers through keyboards and mouse devices. Information also can be entered in many other ways, including by speaking to your computer, scanning images and barcodes, reading from secondary storage devices (like hard drives, CD drives, DVD drives and USB drives—also called “thumb drives”) and having your computer receive information from the Internet (such as when you download videos from YouTube™, e-books from Amazon and the like).

2. **Output unit.** This “shipping” section takes information that the computer has processed and places it on various output devices to make it available for use outside the computer. Most information that is output from computers today is displayed on screens, printed on paper, played on audio players (such as Apple’s popular iPods), or used to control other devices. Computers also can output their information to networks, such as the Internet.

3. **Memory unit.** This rapid-access, relatively low-capacity “warehouse” section retains information that has been entered through the input unit, making it immediately available for processing when needed. The memory unit also retains processed information until it can be placed on output devices by the output unit. Information in the memory unit is volatile—it’s typically lost when the computer’s power is turned off. The memory unit is often called either memory or primary memory.

4. **Arithmetic and logic unit (ALU).** This “manufacturing” section performs calculations, such as addition, subtraction, multiplication and division. It also contains the decision mechanisms that allow the computer, for example, to compare two items from the memory unit to determine whether they’re equal. In today’s systems, the ALU is usually implemented as part of the next logical unit, the CPU.

5. **Central processing unit (CPU).** This “administrative” section coordinates and supervises the operation of the other sections. The CPU tells the input unit when to
read information into the memory unit, tells the ALU when information from the memory unit should be used in calculations and tells the output unit when to send information from the memory unit to certain output devices. Many of today’s computers have multiple CPUs and, hence, can perform many operations simultaneously—such computers are called multiprocessors. A multi-core processor implements multiprocessing on a single integrated circuit chip—for example a dual-core processor has two CPUs and a quad-core processor has four CPUs.

6. **Secondary storage unit.** This is the long-term, high-capacity “warehousing” section. Programs or data not actively being used by the other units normally are placed on secondary storage devices (e.g., your hard drive) until they’re again needed, possibly hours, days, months or even years later. Therefore, information on secondary storage devices is said to be persistent—it is preserved even when the computer’s power is turned off. Secondary storage information takes much longer to access than information in primary memory, but the cost per unit of secondary storage is much less than that of primary memory. Examples of secondary storage devices include CDs, DVDs and flash drives (sometimes called memory sticks), which can hold hundreds of millions to billions of characters.

### 1.4 Personal, Distributed and Client/Server Computing

In 1977, Apple Computer popularized personal computing. Computers became so economical that people could buy them for their own personal or business use. In 1981, IBM, the world’s largest computer vendor, introduced the IBM Personal Computer (PC). This quickly legitimized personal computing in business, industry and government organizations, where IBM mainframes were heavily used.

These computers were “stand-alone” units—people transported disks back and forth between them to share information (this was often called “sneakernet”). These machines could be linked together in computer networks, sometimes over telephone lines and sometimes in local area networks (LANs) within an organization. This led to the phenomenon of distributed computing, in which an organization’s computing, instead of being performed only at some central computer installation, is distributed over networks to the sites where the organization’s work is performed. Personal computers were powerful enough to handle the computing requirements of individual users as well as the basic communications tasks of passing information between computers electronically.

Today’s personal computers are as powerful as the million-dollar machines of just a few decades ago. Information is shared easily across computer networks, where computers called servers (file servers, database servers, web servers, etc.) offer capabilities that may be used by client computers distributed throughout the network, hence the term client/server computing. C is widely used for writing software for operating systems, for computer networking and for distributed client/server applications. Today’s popular operating systems such as UNIX, Linux, Mac OS X and Microsoft’s Windows-based systems provide the kinds of capabilities discussed in this section.

### 1.5 The Internet and the World Wide Web

The Internet—a global network of computers—was initiated in the late 1960s with funding supplied by the U.S. Department of Defense. Originally designed to connect the main
computer systems of about a dozen universities and research organizations, the Internet today is accessible by computers worldwide.

With the introduction of the World Wide Web—which allows computer users to locate and view multimedia-based documents on almost any subject over the Internet—the Internet has exploded into the world’s premier communication mechanism.

The Internet and the World Wide Web are surely among humankind’s most important and profound creations. In the past, most computer applications ran on computers that were not connected to one another. Today’s applications can be written to communicate among the world’s computers. The Internet mixes computing and communications technologies. It makes our work easier. It makes information instantly and conveniently accessible worldwide. It enables individuals and local small businesses to get worldwide exposure. It’s changing the way business is done. People can search for the best prices on virtually any product or service. Special-interest communities can stay in touch with one another. Researchers can be made instantly aware of the latest breakthroughs.

### 1.6 Machine Languages, Assembly Languages and High-Level Languages

Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate translation steps. Hundreds of computer languages are in use today. These may be divided into three general types:

1. Machine languages
2. Assembly languages
3. High-level languages

Any computer can directly understand only its own machine language. Machine language is the “natural language” of a computer and as such is defined by its hardware design. [Note: Machine language is often referred to as object code. This term predates “object-oriented programming.” These two uses of “object” are unrelated.] Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are machine dependent (i.e., a particular machine language can be used on only one type of computer). Such languages are cumbersome for humans, as illustrated by the following section of an early machine-language program that adds overtime pay to base pay and stores the result in gross pay:

```
+1300042774
+1400593419
+1200274027
```

Machine-language programming was simply too slow, tedious and error prone for most programmers. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations. These abbreviations formed the basis of assembly languages. Translator programs called assemblers were developed to convert early assembly-language programs to machine language at computer speeds. The following section of an assembly-language program also adds overtime pay to base pay and stores the result in gross pay:
Although such code is clearer to humans, it’s incomprehensible to computers until translated to machine language.

Computer usage increased rapidly with the advent of assembly languages, but programmers still had to use many instructions to accomplish even the simplest tasks. To speed the programming process, **high-level languages** were developed in which single statements could be written to accomplish substantial tasks. Translator programs called **compilers** convert high-level language programs into machine language. High-level languages allow programmers to write instructions that look almost like everyday English and contain commonly used mathematical notations. A payroll program written in a high-level language might contain a statement such as

\[
grossPay = basePay + overTimePay;
\]

From your standpoint, obviously, high-level languages are preferable to machine and assembly language. C, C++, Microsoft’s .NET languages (e.g., Visual Basic, Visual C++ and Visual C#) and Java are among the most widely used high-level programming languages.

The process of compiling a high-level language program into machine language can take a considerable amount of computer time. **Interpreter** programs were developed to execute high-level language programs directly (without the delay of compilation), although slower than compiled programs run.

### 1.7 History of C

C evolved from two previous languages, BCPL and B. BCPL was developed in 1967 by Martin Richards as a language for writing operating-systems software and compilers. Ken Thompson modeled many features in his B language after their counterparts in BCPL, and in 1970 he used B to create early versions of the UNIX operating system at Bell Laboratories. Both BCPL and B were “typeless” languages—every data item occupied one “word” in memory, and the burden of typing variables fell on the shoulders of the programmer.

The C language was evolved from B by Dennis Ritchie at Bell Laboratories and was originally implemented on a DEC PDP-11 computer in 1972. C uses many of the important concepts of BCPL and B while adding data typing and other powerful features. C initially became widely known as the development language of the UNIX operating system. Today, virtually all new major operating systems are written in C and/or C++. C is available for most computers. C is mostly hardware independent. With careful design, it’s possible to write C programs that are **portable** to most computers.

By the late 1970s, C had evolved into what is now referred to as “traditional C.” The publication in 1978 of Kernighan and Ritchie’s book, *The C Programming Language*, drew wide attention to the language. This became one of the most successful computer science books of all time.

The rapid expansion of C over various types of computers (sometimes called **hardware platforms**) led to many variations that were similar but often incompatible. This was a serious problem for programmers who needed to develop code that would run on several platforms. It became clear that a standard version of C was needed. In 1983, the X3J11
Chapter 1  Introduction to Computers, the Internet and the Web

technical committee was created under the American National Standards Committee on Computers and Information Processing (X3) to “provide an unambiguous and machine-independent definition of the language.” In 1989, the standard was approved; this standard was updated in 1999. The standards document is referred to as INCITS/ISO/IEC 9899-1999. Copies may be ordered from the American National Standards Institute (www.ansi.org) at webstore.ansi.org/ansidocstore.

C99 is a revised standard for the C programming language that refines and expands the capabilities of C. Not all popular C compilers support C99. Of those that do, most implement only a subset of the new features. Chapters 1–14 of this book are based on the widely adopted international Standard (ANSI/ISO) C. Appendix G introduces C99 and provides links to popular C99 compilers and IDEs.

1.8 C Standard Library

As you’ll learn in Chapter 5, C programs consist of modules or pieces called functions. You can program all the functions you need to form a C program, but most C programmers take advantage of a rich collection of existing functions called the C Standard Library. Thus, there are really two pieces to learning how to program in C. The first is learning the C language itself, and the second is learning how to use the functions in the C Standard Library. Throughout the book, we discuss many of these functions. P.J. Plauger’s book The Standard C Library is must reading for programmers who need a deep understanding of the library functions, how to implement them and how to use them to write portable code. We use and explain many C library functions throughout this text. Visit the following website for the complete C Standard Library documentation, including the C99 features:

www.dinkumware.com/manuals/default.aspx#Standard%20C%20Library

This textbook encourages a building-block approach to creating programs. Avoid reinventing the wheel. Instead, use existing pieces—this is called software reusability, and it’s a key to the field of object-oriented programming, as you’ll see in our treatment of C++ beginning in Chapter 15. When programming in C you’ll typically use the following building blocks:

• C Standard Library functions

• Functions you create yourself

• Functions other people have created and made available to you

The advantage of creating your own functions is that you’ll know exactly how they work. You’ll be able to examine the C code. The disadvantage is the time-consuming effort that goes into designing, developing and debugging new functions.

If you use existing functions, you can avoid reinventing the wheel. In the case of the Standard C functions, you know that they’re carefully written, and you know that because you’re using functions that are available on virtually all Standard C implementations, your programs will have a greater chance of being portable and error-free.
1.9 C++

C++ was developed by Bjarne Stroustrup at Bell Laboratories. It has its roots in C, providing a number of features that “spruce up” the C language. More important, it provides capabilities for object-oriented programming. C++ has become a dominant language in both industry and the colleges.

Objects are essentially reusable software components that model items in the real world. Using a modular, object-oriented design and implementation approach can make software development groups much more productive than is possible with previous programming techniques.

Many people feel that the best educational strategy today is to master C, then study C++. Therefore, in Chapters 15–24 of *C How to Program, 6/e*, we present a condensed treatment of C++ selected from our book *C++ How to Program, 7/e*. As you study C++, check out our online C++ Resource Center at www.deitel.com/cplusplus/.

1.10 Java

Microprocessors are having a profound impact in intelligent consumer electronic devices. Recognizing this, Sun Microsystems in 1991 funded an internal corporate research project code-named Green. The project resulted in the development of a C++-based language that its creator, James Gosling, called Oak after an oak tree outside his window at Sun. It was later discovered that there already was a computer language called Oak. When a group of Sun people visited a local coffee shop, the name Java was suggested and it stuck.

The Green project ran into some difficulties. The marketplace for intelligent consumer electronic devices did not develop in the early 1990s as quickly as Sun had anticipated. The project was in danger of being canceled. By sheer good fortune, the World Wide Web exploded in popularity in 1993, and Sun saw the immediate potential of using Java to add dynamic content (e.g., interactivity, animations and the like) to web pages. This breathed new life into the project.

Sun formally announced Java at an industry conference in May 1995. Java garnered the attention of the business community because of the phenomenal interest in the World Wide Web. Java is now used to develop large-scale enterprise applications, to enhance the functionality of web servers (the computers that provide the content we see in our web browsers), to provide applications for consumer devices (e.g., cell phones, pagers and personal digital assistants) and for many other purposes.
### 1.11 Fortran, COBOL, Pascal and Ada

Hundreds of high-level languages have been developed, but few have achieved broad acceptance. **FORTRAN** (FORmula TRANslator) was developed by IBM Corporation in the mid-1950s to be used for scientific and engineering applications that require complex mathematical computations. Fortran is still widely used in engineering applications.

**COBOL** (COmmon Business Oriented Language) was developed in the late 1950s by computer manufacturers, the U.S. government and industrial computer users. COBOL is used for commercial applications that require precise and efficient manipulation of large amounts of data. Much business software is still programmed in COBOL.

During the 1960s, many large software development efforts encountered severe difficulties. Software deliveries were often late, costs greatly exceeded budgets and the finished products were unreliable. People realized that software development was a more complex activity than they had imagined. Research in the 1960s resulted in the evolution of **structured programming**—a disciplined approach to writing programs that are clearer and easier to test, debug and modify than large programs produced with previous techniques.

One of the more tangible results of this research was the development of the **Pascal** programming language by Professor Niklaus Wirth in 1971. Named after the seventeenth-century mathematician and philosopher Blaise Pascal, it was designed for teaching structured programming and rapidly became the preferred programming language in most colleges. Pascal lacked many features needed in commercial, industrial and government applications, so it was not widely accepted outside academia.

The **Ada** language was developed under the sponsorship of the U.S. Department of Defense (DoD) during the 1970s and early 1980s. Hundreds of separate languages were being used to produce the DoD’s massive command-and-control software systems. The DoD wanted one language that would fill most of its needs. The Ada language was named after Lady Ada Lovelace, daughter of the poet Lord Byron. Lady Lovelace is credited with writing the world’s first computer program in the early 1800s (for the Analytical Engine mechanical computing device designed by Charles Babbage). One important capability of Ada, called **multitasking**, allows programmers to specify that many activities are to occur in parallel. Although multithreading is not part of standard C, it’s available through various add-on libraries.

### 1.12 BASIC, Visual Basic, Visual C++, C# and .NET

The **BASIC** (Beginner’s All-purpose Symbolic Instruction Code) programming language was developed in the mid-1960s at Dartmouth College as a means of writing simple programs. BASIC’s primary purpose was to familiarize novices with programming techniques. Microsoft’s **Visual Basic** language, introduced in the early 1990s to simplify the development of Microsoft Windows applications, has become one of the most popular programming languages in the world.

Microsoft’s latest development tools are part of its corporate-wide strategy for integrating the Internet and the web into computer applications. This strategy is implemented in Microsoft’s **.NET platform**, which provides the capabilities developers need to create and run computer applications that can execute on computers distributed across the Internet. Microsoft’s three primary programming languages are **Visual Basic** (based on the original BASIC), **Visual C++** (based on C++) and **Visual C#** (a new language based on
C++ and Java that was developed expressly for the .NET platform). Visual C++ can also be used to compile and run C programs.

### 1.13 Key Software Trend: Object Technology

One of the authors, Harvey Deitel, remembers the great frustration felt in the 1960s by software development organizations, especially those working on large-scale projects. During his undergraduate years at MIT, he worked summers at a leading computer vendor on the teams developing timesharing, virtual memory operating systems. This was a great experience for a college student. But, in the summer of 1967, reality set in when the company “decommitted” from producing as a commercial product the particular system on which hundreds of people had been working for many years. It was difficult to get this software right—software is “complex stuff.”

Improvements to software technology did emerge, with the benefits of structured programming (and the related disciplines of structured systems analysis and design) being realized in the 1970s. Not until the technology of object-oriented programming became widely used in the 1990s, though, did software developers feel they had the necessary tools for making major strides in the software development process.

Actually, object technology dates back to the mid 1960s. The C++ programming language, developed at AT&T by Bjarne Stroustrup in the early 1980s, is based on two languages—C and Simula 67, a simulation programming language developed at the Norwegian Computing Center and released in 1967. C++ absorbed the features of C and added Simula’s capabilities for creating and manipulating objects. Neither C nor C++ was originally intended for wide use beyond the AT&T research laboratories. But grass roots support rapidly developed for each.

Object technology is a packaging scheme that helps us create meaningful software units. There are date objects, time objects, paycheck objects, invoice objects, audio objects, video objects, file objects, record objects and so on. In fact, almost any noun can be reasonably represented as an object.

We live in a world of objects. There are cars, planes, people, animals, buildings, traffic lights, elevators and the like. Before object-oriented languages appeared, procedural programming languages (such as Fortran, COBOL, Pascal, BASIC and C) were focused on actions (verbs) rather than on things or objects (nouns). Programmers living in a world of objects programmed primarily using verbs. This made it awkward to write programs. Now, with the availability of popular object-oriented languages such as C++, Java and C#, programmers continue to live in an object-oriented world and can program in an object-oriented manner. This is a more natural process than procedural programming and has resulted in significant productivity gains.

A key problem with procedural programming is that the program units do not effectively mirror real-world entities, so these units are not particularly reusable. It isn’t unusual for programmers to “start fresh” on each new project and have to write similar software “from scratch.” This wastes time and money, as people repeatedly “reinvent the wheel.” With object technology, the software entities created (called classes), if properly designed, tend to be reusable on future projects. Using libraries of reusable componentry can greatly reduce effort required to implement certain kinds of systems (compared to the effort that would be required to reinvent these capabilities on new projects).
Some organizations report that the key benefit of object-oriented programming is not software reuse but, rather, that the software they produce is more understandable, better organized and easier to maintain, modify and debug. This can be significant, because perhaps as much as 80 percent of software costs are associated not with the original efforts to develop the software, but with the continued evolution and maintenance of that software throughout its lifetime. Whatever the perceived benefits, it’s clear that object-oriented programming will be the key programming methodology for the next several decades.

1.14 Typical C Program Development Environment

C systems generally consist of several parts: a program development environment, the language and the C Standard Library. The following discussion explains the typical C development environment shown in Fig. 1.1.

C programs typically go through six phases to be executed (Fig. 1.1). These are: edit, preprocess, compile, link, load and execute. Although *C How to Program, 6/e* is a generic C textbook (written independently of the details of any particular operating system), we concentrate in this section on a typical Linux-based C system. [Note: The programs in this book will run with little or no modification on most current C systems, including Microsoft Windows-based systems.] If you’re not using a Linux system, refer to the manuals for your system or ask your instructor how to accomplish these tasks in your environment. Check out our C Resource Center at [www.deitel.com/C](http://www.deitel.com/C) to locate “getting started” tutorials for popular C compilers and development environments.

**Phase 1: Creating a Program**

Phase 1 consists of editing a file. This is accomplished with an editor program. Two editors widely used on Linux systems are *vi* and *emacs*. Software packages for the C/C++ integrated program development environments such as Eclipse and Microsoft Visual Studio have editors that are integrated into the programming environment. You type a C program with the editor, make corrections if necessary, then store the program on a secondary storage device such as a hard disk. C program file names should end with the `.c` extension.

**Phases 2 and 3: Preprocessing and Compiling a C Program**

In Phase 2, the you give the command to compile the program. The compiler translates the C program into machine language-code (also referred to as object code). In a C system, a preprocessor program executes automatically before the compiler’s translation phase begins. The C preprocessor obeys special commands called preprocessor directives, which indicate that certain manipulations are to be performed on the program before compilation. These manipulations usually consist of including other files in the file to be compiled and performing various text replacements. The most common preprocessor directives are discussed in the early chapters; a detailed discussion of preprocessor features appears in Chapter 13. In Phase 3, the compiler translates the C program into machine-language code.
Phase 4: Linking

The next phase is called linking. C programs typically contain references to functions defined elsewhere, such as in the standard libraries or in the private libraries of groups of programmers working on a particular project. The object code produced by the C compiler typically contains “holes” due to these missing parts. A linker links the object code with...
the code for the missing functions to produce an executable image (with no missing pieces). On a typical Linux system, the command to compile and link a program is called cc (or gcc). To compile and link a program named welcome.c type

```
cc welcome.c
```
at the Linux prompt and press the Enter key (or Return key). [Note: Linux commands are case sensitive; make sure that you type lowercase c’s and that the letters in the filename are in the appropriate case.] If the program compiles and links correctly, a file called a.out is produced. This is the executable image of our welcome.c program.

**Phase 5: Loading**
The next phase is called loading. Before a program can be executed, the program must first be placed in memory. This is done by the loader, which takes the executable image from disk and transfers it to memory. Additional components from shared libraries that support the program are also loaded.

**Phase 6: Execution**
Finally, the computer, under the control of its CPU, executes the program one instruction at a time. To load and execute the program on a Linux system, type ./a.out at the Linux prompt and press Enter.

**Problems That May Occur at Execution Time**
Programs do not always work on the first try. Each of the preceding phases can fail because of various errors that we’ll discuss. For example, an executing program might attempt to divide by zero (an illegal operation on computers just as in arithmetic). This would cause the computer to display an error message. You would then return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine that the corrections work properly.

Most C programs input and/or output data. Certain C functions take their input from stdin (the standard input stream), which is normally the keyboard, but stdin can be connected to another stream. Data is often output to stdout (the standard output stream), which is normally the computer screen, but stdout can be connected to another stream. When we say that a program prints a result, we normally mean that the result is displayed on a screen. Data may be output to devices such as disks and printers. There is also a standard error stream referred to as stderr. The stderr stream (normally connected to the screen) is used for displaying error messages. It’s common to route regular output data, i.e., stdout, to a device other than the screen while keeping stderr assigned to the screen so that the user can be immediately informed of errors.

**Common Programming Error 1.1**
Errors like division-by-zero occur as a program runs, so these errors are called runtime errors or execution-time errors. Divide-by-zero is generally a fatal error, i.e., an error that causes the program to terminate immediately without successfully performing its job. Non-fatal errors allow programs to run to completion, often producing incorrect results.

**1.15 Hardware Trends**
Every year, people generally expect to pay at least a little more for most products and services. The opposite has been the case in the computer and communications fields, espe-
cially with regard to the costs of hardware supporting these technologies. For many decades, hardware costs have fallen rapidly, if not precipitously. Every year or two, the capacities of computers have approximately doubled without any increase in price. This often is called Moore’s Law, named after the person who first identified and explained the trend, Gordon Moore, cofounder of Intel—the company that manufactures the vast majority of the processors in today’s personal computers. Moore’s Law and similar trends are especially true in relation to the amount of memory that computers have for programs, the amount of secondary storage (such as disk storage) they have to hold programs and data over longer periods of time, and their processor speeds—the speeds at which computers execute programs (i.e., do their work). Similar growth has occurred in the communications field, in which costs have plummeted as soaring demand for communications bandwidth has attracted intense competition. We know of no other fields in which technology improves so quickly and costs fall so rapidly. Such improvement in the computing and communications fields is truly fostering the so-called Information Revolution.

1.16 Notes About C and This Book

Experienced C programmers sometimes take pride in creating weird, contorted, convoluted usages of the language. This is a poor programming practice. It makes programs more difficult to read, more likely to behave strangely, more difficult to test and debug and more difficult to adapt to changing requirements. This book is geared for novice programmers, so we stress program clarity. The following is our first “good programming practice.”

**Good Programming Practice 1.1**

Write your C programs in a simple and straightforward manner. This is sometimes referred to as KIS (“keep it simple”). Do not “stretch” the language by trying bizarre usages.

You may have heard that C is a portable language and that programs written in C can run on many different computers. Portability is an elusive goal. The Standard C document contains a lengthy list of portability issues, and complete books have been written that discuss portability.

**Portability Tip 1.2**

Although it’s possible to write portable C programs, there are many problems between different C compilers and different computers that make portability difficult to achieve. Simply writing programs in C does not guarantee portability. You’ll often need to deal directly with computer variations.

We’ve done a careful walkthrough of the C Standard and audited our presentation against it for completeness and accuracy. However, C is a rich language, and there are some subtleties in the language and some advanced subjects we have not covered. If you need additional technical details on C, we suggest that you read the C Standard document itself or the book by Kernighan and Ritchie (The C Programming Language, Second Edition).

**Software Engineering Observation 1.2**

Read the manuals for the version of C you’re using. Reference these manuals frequently to be sure you’re aware of the rich collection of C features and that you’re using these features correctly.
Software Engineering Observation 1.3

Your computer and compiler are good teachers. If you’re not sure how a C feature works, write a program with that feature, compile and run the program and see what happens.

1.17 Web Resources

This section provides links to our C and related Resource Centers that will be useful to you as you learn C. These Resource Centers include various C resources, including, blogs, articles, whitepapers, compilers, development tools, downloads, FAQs, tutorials, webcasts, wikis and links to resources for C game programming with the Allegro libraries.

*Deitel & Associates Websites*

www.deitel.com/books/chtp6/
The Deitel & Associates C How to Program, 6/e site. Here you’ll find links to the book’s examples and other resources.

www.deitel.com/C/

www.deitel.com/visualcplusplus/

www.deitel.com/codesearchengines/

www.deitel.com/programmingprojects/

Check these Resource Centers for compilers, code downloads, tutorials, documentation, books, e-books, articles, blogs, RSS feeds and more that will help you develop C applications.

www.deitel.com

Check this site for updates, corrections and additional resources for all Deitel publications.

www.deitel.com/newsletter/subscribe.html

Subscribe here for the Deitel® Buzz Online e-mail newsletter to follow the Deitel & Associates publishing program, including updates and errata to C How to Program, 6/e.

Summary

**Section 1.1 Introduction**

- Software (i.e., the instructions you write to command computers to perform actions and make decisions) controls computers (often referred to as hardware).
- C was standardized in 1989 in the United States through the American National Standards Institute (ANSI) then worldwide through the International Standards Organization (ISO).
- Silicon-chip technology has made computing so economical that more than a billion general-purpose computers are in use worldwide.

**Section 1.2 Computers: Hardware and Software**

- A computer is capable of performing computations and making logical decisions at speeds billions of times faster than human beings can.
- Computers process data under the control of sets of instructions called computer programs, which guide the computer through orderly sets of actions specified by computer programmers.
- The various devices that comprise a computer system are referred to as hardware.
- The computer programs that run on a computer are referred to as software.

**Section 1.3 Computer Organization**

- The input unit is the “receiving” section of the computer. It obtains information from input devices and places it at the disposal of the other units for processing.
• The output unit is the “shipping” section of the computer. It takes information processed by the computer and places it on output devices to make it available for use outside the computer.

• The memory unit is the rapid-access, relatively low-capacity “warehouse” section of the computer. It retains information that has been entered through the input unit, making it immediately available for processing when needed, and retains information that has already been processed until it can be placed on output devices by the output unit.

• The arithmetic and logic unit (ALU) is the “manufacturing” section of the computer. It’s responsible for performing calculations and making decisions.

• The central processing unit (CPU) is the “administrative” section of the computer. It coordinates and supervises the operation of the other sections.

• The secondary storage unit is the long-term, high-capacity “warehousing” section of the computer. Programs or data not being used by the other units are normally placed on secondary storage devices (e.g., disks) until they’re needed, possibly hours, days, months or even years later.

Section 1.4 Personal, Distributed and Client/Server Computing
• Apple Computer popularized personal computing.

• IBM’s Personal Computer quickly legitimized personal computing in business, industry and government organizations, where IBM mainframes are heavily used.

• Early personal computers could be linked together in computer networks. This led to the phenomenon of distributed computing.

• Information is shared easily across networks, where computers called servers (file servers, database servers, web servers, etc.) offer capabilities that may be used by client computers distributed throughout the network, hence the term client/server computing.

• C has become widely used for writing software for operating systems, for computer networking and for distributed client/server applications.

Section 1.5 The Internet and the World Wide Web
• The Internet—a global network of computers—was initiated almost four decades ago with funding supplied by the U.S. Department of Defense.

• With the introduction of the World Wide Web—which allows computer users to locate and view multimedia-based documents on almost any subject over the Internet—the Internet has exploded into the world’s premier communication mechanism.

Section 1.6 Machine Languages, Assembly Languages and High-Level Languages
• Any computer can directly understand only its own machine language, which generally consists of strings of numbers that instruct computers to perform their most elementary operations.

• English-like abbreviations form the basis of assembly languages. Translator programs called assemblers convert assembly-language programs to machine language.

• Compilers translate high-level language programs into machine-language programs. High-level languages (like C) contain English words and conventional mathematical notations.

• Interpreter programs directly execute high-level language programs, eliminating the need to compile them into machine language.

Section 1.7 History of C
• C evolved from two previous languages, BCPL and B. BCPL was developed in 1967 by Martin Richards as a language for writing operating systems software and compilers. Ken Thompson modeled many features in his B language after their counterparts in BCPL and used B to create early versions of the UNIX operating system.
Chapter 1 Introduction to Computers, the Internet and the Web

• The C language was evolved from B by Dennis Ritchie at Bell Laboratories. C uses many of the important concepts of BCPL and B while adding data typing and other powerful features.
• C initially became widely known as the development language of the UNIX operating system.
• C is available for most computers. C is mostly hardware independent.
• In 1989, the C standard was approved; this standard was updated in 1999. The standards document is referred to as INCITS/ISO/IEC 9899-1999.
• C99 is a revised standard for the C programming language that refines and expands the capabilities of C, but it has not be universally adopted.

Section 1.8 C Standard Library
• When programming in C you’ll typically use C Standard Library functions, functions you create yourself and functions other people have created and made available to you.

Section 1.9 C++
• C++ was developed by Bjarne Stroustrup at Bell Laboratories. It has its roots in C and provides capabilities for object-oriented programming.
• Objects are essentially reusable software components that model items in the real world.
• Using a modular, object-oriented design and implementation approach makes software development groups much more productive than is possible with conventional programming techniques.

Section 1.10 Java
• Java is used to create dynamic and interactive content for web pages, develop enterprise applications, enhance web server functionality, provide applications for consumer devices and more.

Section 1.11 Fortran, COBOL, Pascal and Ada
• FORTRAN was developed by IBM Corporation in the 1950s for scientific and engineering applications that require complex mathematical computations.
• COBOL was developed in the 1950s for commercial applications that require precise and efficient data manipulation.
• Pascal was designed for teaching structured programming.
• Ada was developed under the sponsorship of the United States Department of Defense (DoD) during the 1970s and early 1980s. Ada provides multitasking, which allows programmers to specify that many activities are to occur in parallel.

Section 1.12 BASIC, Visual Basic, Visual C++, C# and .NET
• BASIC was developed in the 1960s at Dartmouth College for programming novices.
• Visual Basic was introduced in the 1990s to simplify developing Windows applications.
• Microsoft has a corporate-wide strategy for integrating the Internet and the web into computer applications. This strategy is implemented in Microsoft’s .NET platform.
• The .NET platform’s three primary programming languages are Visual Basic (based on the original BASIC), Visual C++ (based on C++) and Visual C# (a new language based on C++ and Java that was developed expressly for the .NET platform).

Section 1.13 Key Software Trend: Object Technology
• Not until object-oriented programming became widely used in the 1990s did software developers feel they had the tools to make major strides in the software development process.
• C++ absorbed the features of C and added Simula’s object capabilities.
• Object technology is a packaging scheme that helps us create meaningful software units.
• With object technology, the software entities created (called classes), if properly designed, tend to be reusable on future projects.
• Some organizations report the key benefit of object-oriented programming is the production of software which is more understandable, better organized and easier to maintain and debug.

Section 1.14 Typical C Program Development Environment
• You create a program by editing a file with an editor program. Software packages for the C/C++ integrated program development environments such as Eclipse and Microsoft Visual Studio have editors that are integrated into the programming environment.
• C program file names should end with the .c extension.
• Compilers translate programs into machine-language code (also referred to as object code).
• A preprocessor program executes automatically before the compiler’s translation phase begins. The C preprocessor obeys special commands called preprocessor directives that usually consist of including other files in the file to be compiled and performing various text replacements.
• A linker links object code with the code for library functions to produce an executable image.
• Before a program can execute, it must first be placed in memory. This is done by the loader. Additional components from shared libraries that support the program are also loaded.
• The computer, under the control of its CPU, executes a program one instruction at a time.

Section 1.15 Hardware Trends
• Every year, people generally expect to pay at least a little more for most products and services. The opposite has been the case in the computer and communications fields, especially with regard to the costs of hardware supporting these technologies. For many decades, hardware costs have fallen rapidly, if not precipitously.
• Every year or two, the capacities of computers have approximately doubled without any increase in price. This often is called Moore’s Law, named after the person who first identified and explained the trend, Gordon Moore, cofounder of Intel—the company that manufactures the vast majority of the processors in today’s personal computers.
• Moore’s Law is especially true in relation to the amount of memory that computers have for programs, the amount of secondary storage they have to hold programs and data over longer periods of time, and their processor speeds—the speeds at which computers execute their programs.

Terminology
actions (computers perform) 2
Ada programming language 10
Allegro 2
American National Standards Institute (ANSI) 2
arithmetic and logic unit (ALU) 4
assembler 6
assembly language 6
BASIC (Beginner’s All-Purpose Symbolic Instruction Code) 10
building-block approach 8
C preprocessor 12
C Standard Library 8
c c compilation command 14
central processing unit (CPU) 4
class 11
client computer 5
client/server computing 5
COBOL (COMmon Business Oriented Language) 10
compile 12
compile phase 12
compiler 7
components (software) 9
computer 3
computer program 4
computer programmer 4
data 4
decisions (made by computers) 2
Self-Review Exercises

1.1 Fill in the blanks in each of the following:
   a) The company that popularized personal computing was ________.
   b) The computer that made personal computing legitimate in business and industry was the ________.
   c) Computers process data under the control of sets of instructions called computer ________.
   d) The six key logical units of the computer are the ________, ________, ________, ________, ________, and the ________.
   e) The three types of languages we discussed are ________, ________, and ________.
   f) The programs that translate high-level language programs into machine language are called ________.
   g) C is widely known as the development language of the ________ operating system.
   h) The Department of Defense developed the Ada language with a capability called ________, which allows programmers to specify activities that can proceed in parallel.
1.2 Fill in the blanks in each of the following sentences about the C environment.
   a) C programs are normally typed into a computer using a(n) ________ program.
   b) In a C system, a(n) ________ program automatically executes before the translation phase begins.
   c) The two most common kinds of preprocessor directives are ________ and ________.
   d) The ________ program combines the output of the compiler with various library functions to produce an executable image.
   e) The ________ program transfers the executable image from disk to memory.
   f) To load and execute the most recently compiled program on a Linux system, type ________.

Answers to Self-Review Exercises
1.1 a) Apple. b) IBM Personal Computer. c) programs. d) input unit, output unit, memory unit, arithmetic and logic unit, central processing unit, secondary storage unit. e) machine languages, assembly languages and high-level languages. f) compilers. g) UNIX. h) multitasking.
1.2 a) editor. b) preprocessor. c) including other files in the file to be compiled, replacing special symbols with program text. d) linker. e) loader. f) ./a.out.

Exercises
1.3 Categorize each of the following items as either hardware or software:
   a) CPU
   b) C++ compiler
   c) ALU
   d) C++ preprocessor
   e) input unit
   f) an editor program

1.4 Why might you want to write a program in a machine-independent language instead of a machine-dependent language? Why might a machine-dependent language be more appropriate for writing certain types of programs?

1.5 Fill in the blanks in each of the following statements:
   a) Which logical unit of the computer receives information from outside the computer for use by the computer? ________.
   b) The process of instructing the computer to solve specific problems is called ________.
   c) What type of computer language uses English-like abbreviations for machine-language instructions? ________.
   d) Which logical unit of the computer sends information that has already been processed by the computer to various devices so that the information may be used outside the computer? ________.
   e) Which logical units of the computer retain information? ________.
   f) Which logical unit of the computer performs calculations? ________.
   g) Which logical unit of the computer makes logical decisions? ________.
   h) The level of computer language most convenient for you to write programs quickly and easily is ________.
   i) The only language that a computer directly understands is called that computer’s ________.
   j) Which logical unit of the computer coordinates the activities of all the other logical units? ________.
1.6 State whether each of the following is true or false. If false, explain your answer.
   a) Machine languages are generally machine dependent.
   b) Like other high-level languages, C is generally considered to be machine independent.

1.7 Discuss the meaning of each of the following names:
   a) stdin
   b) stdout
   c) stderr

1.8 Why is so much attention today focused on object-oriented programming?

1.9 Which programming language is best described by each of the following?
   a) Developed by IBM for scientific and engineering applications.
   b) Developed specifically for business applications.
   c) Developed for teaching structured programming.
   d) Named after the world’s first computer programmer.
   e) Developed to familiarize novices with programming techniques.
   f) Specifically developed to help programmers migrate to .NET.
   g) Known as the development language of UNIX.
   h) Formed primarily by adding object-oriented programming to C.
   i) Succeeded initially because of its ability to create web pages with dynamic content.

Making a Difference

1.10 (Test-Drive: Carbon Footprint Calculator) Some scientists believe that carbon emissions, especially from the burning of fossil fuels, contribute significantly to global warming and that this can be combated if individuals take steps to limit their use of carbon-based fuels. Organizations and individuals are increasingly concerned about their “carbon footprints.” Websites such as TerraPass
   www.terrapass.com/carbon-footprint-calculator/
   and Carbon Footprint
   www.carbonfootprint.com/calculator.aspx
   provide carbon footprint calculators. Test-drive these calculators to estimate your carbon footprint. Exercises in later chapters will ask you to program your own carbon footprint calculator. To prepare for this, use the web to research the formulas for calculating carbon footprints.

1.11 (Test-Drive: Body Mass Index Calculator) By recent estimates, two-thirds of the people in the United States are overweight and about half of those are obese. This causes significant increases in illnesses such as diabetes and heart disease. To determine whether a person is overweight or obese, you can use a measure called the body mass index (BMI). The United States Department of Health and Human Services provides a BMI calculator at www.nhlbisupport.com/bmi/. Use it to calculate your own BMI. An exercise in Chapter 2 will ask you to program your own BMI calculator. To prepare for this, use the web to research the formulas for calculating BMI.

1.12 (Gender Neutrality) Many people want to eliminate sexism in all forms of communication. You’ve been asked to create a program that can process a paragraph of text and replace gender-specific words with gender-neutral ones. Assuming that you’ve been given a list of gender-specific words and their gender-neutral replacements (e.g., replace “wife” with “spouse,” “man” with “person,” “daughter” with “child” and so on), explain the procedure you’d use to read through a paragraph of text and manually perform these replacements. How might your procedure generate a strange term like “wope rchild,” which is actually listed in the Urban Dictionary (www.urbandictionary.com)? In Chapter 4, you’ll learn that a more formal term for “procedure” is “algorithm,” and that an algorithm specifies the steps to be performed and the order in which to perform them.
What’s in a name? That which we call a rose
By any other name would smell as sweet.
—William Shakespeare

When faced with a decision, I always ask, “What would be the most fun?”
—Peggy Walker

“Take some more tea,” the March Hare said to Alice, very earnestly. “I’ve had nothing yet,” Alice replied in an offended tone: “so I can’t take more.” “You mean you can’t take less,” said the Hatter: “it’s very easy to take more than nothing.”
—Lewis Carroll

High thoughts must have high language.
—Aristophanes

Objectives
In this chapter, you’ll learn

■ To write simple computer programs in C.
■ To use simple input and output statements.
■ To use the fundamental data types.
■ Computer memory concepts.
■ To use arithmetic operators.
■ The precedence of arithmetic operators.
■ To write simple decision-making statements.
### 2.1 Introduction

The C language facilitates a structured and disciplined approach to computer program design. In this chapter we introduce C programming and present several examples that illustrate many important features of C. Each example is analyzed one statement at a time. In Chapters 3 and 4 we present an introduction to structured programming in C. We then use the structured approach throughout the remainder of the C portion of the text.

### 2.2 A Simple C Program: Printing a Line of Text

C uses some notations that may appear strange to people who have not programmed computers. We begin by considering a simple C program. Our first example prints a line of text. The program and its screen output are shown in Fig. 2.1.

```c
/* Fig. 2.1: fig02_01.c
A first program in C */
#include <stdio.h>
/* function main begins program execution */
int main( void )
{
    printf( "Welcome to C!\n" );
    return 0; /* indicate that program ended successfully */
} /* end function main */
```

**Fig. 2.1 | A first program in C.**

Even though this program is simple, it illustrates several important features of the C language. Lines 1 and 2 begin with /* and end with */ indicating that these two lines are a comment. You insert comments to document programs and improve program readability. Comments do not cause the computer to perform any action when the program is run. Comments are ignored by the C compiler and do not cause any machine-language object code to be
generated. The preceding comment simply describes the figure number, file name and purpose of the program. Comments also help other people read and understand your program, but too many comments can make a program difficult to read.

**Common Programming Error 2.1**
*Forgetting to terminate a comment with */.*

**Common Programming Error 2.2**
*Starting a comment with the characters */ or ending a comment with the characters */.*

C99 also includes the C++ language’s // single-line comments in which everything from // to the end of the line is a comment. These can be used as standalone comments on lines by themselves or as end-of-line comments to the right of a partial line of code. Some programmers prefer // comments because they’re shorter and they eliminate the common programming errors that occur with /* */ comments.

Line 3

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

is a directive to the C preprocessor. Lines beginning with # are processed by the preprocessor before the program is compiled. Line 3 tells the preprocessor to include the contents of the standard input/output header (<stdio.h>) in the program. This header contains information used by the compiler when compiling calls to standard input/output library functions such as printf. We explain the contents of headers in more detail in Chapter 5.

Line 6

```c
int main( void )
```

is a part of every C program. The parentheses after main indicate that main is a program building block called a function. C programs contain one or more functions, one of which must be main. Every program in C begins executing at the function main. Functions can return information. The keyword int to the left of main indicates that main “returns” an integer (whole number) value. We’ll explain what it means for a function to “return a value” when we demonstrate how to create your own functions in Chapter 5. For now, simply include the keyword int to the left of main in each of your programs. Functions also can receive information when they’re called upon to execute. The void in parentheses here means that main does not receive any information. In Chapter 14, Other C Topics, we’ll show an example of main receiving information.

**Good Programming Practice 2.1**
*Every function should be preceded by a comment describing the purpose of the function.*

A left brace, {, begins the body of every function (line 7). A corresponding right brace ends each function (line 11). This pair of braces and the portion of the program between the braces is called a block. The block is an important program unit in C.

Line 8

```c
printf( "Welcome to C!\n" );
```
instructs the computer to perform an **action**, namely to print on the screen the **string** of characters marked by the quotation marks. A string is sometimes called a **character string**, a **message** or a **literal**. The entire line, including `printf`, its **argument** within the parentheses and the semicolon (;), is called a **statement**. Every statement must end with a semicolon (also known as the **statement terminator**). When the preceding `printf` statement is executed, it prints the message **Welcome to C!** on the screen. The characters normally print exactly as they appear between the double quotes in the `printf` statement. Notice that the characters `\n` were not printed on the screen. The backslash (`\`) is called an **escape character**. It indicates that `printf` is supposed to do something out of the ordinary. When encountering a backslash in a string, the compiler looks ahead at the next character and combines it with the backslash to form an **escape sequence**. The escape sequence `\n` means **newline**. When a newline appears in the string output by a `printf`, the newline causes the cursor to position to the beginning of the next line on the screen. Some common escape sequences are listed in Fig. 2.2.

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\n</code></td>
<td>Newline. Position the cursor at the beginning of the next line.</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>Horizontal tab. Move the cursor to the next tab stop.</td>
</tr>
<tr>
<td><code>\a</code></td>
<td>Alert. Sound the system bell.</td>
</tr>
<tr>
<td><code>\</code></td>
<td>Backslash. Insert a backslash character in a string.</td>
</tr>
<tr>
<td><code>\&quot;</code></td>
<td>Double quote. Insert a double-quote character in a string.</td>
</tr>
</tbody>
</table>

**Fig. 2.2** | Some common escape sequences.

The last two escape sequences in Fig. 2.2 may seem strange. Because the backslash has special meaning in a string, i.e., the compiler recognizes it as an escape character, we use a double backslash (`\`) to place a single backslash in a string. Printing a double quote also presents a problem because double quotes mark the boundary of a string—such quotes are not printed. By using the escape sequence `\"` in a string to be output by `printf`, we indicate that `printf` should display a double quote.

Line 10

```c
return 0; /* indicate that program ended successfully */
```

is included at the end of every **main** function. The keyword `return` is one of several means we’ll use to **exit a function**. When the return statement is used at the end of **main** as shown here, the value 0 indicates that the program has terminated successfully. In Chapter 5 we discuss functions in detail, and the reasons for including this statement will become clear. For now, simply include this statement in each program, or the compiler might produce a warning on some systems. The right brace, `}`, (line 12) indicates that the end of **main** has been reached.

**Good Programming Practice 2.2**

*Add a comment to the line containing the right brace, `}`, that closes every function, including **main**.*
We said that `printf` causes the computer to perform an action. As any program executes, it performs a variety of actions and makes **decisions**. At the end of this chapter, we discuss decision making. In Chapter 3, we discuss this **action/decision model** of programming in depth.

Standard library functions like `printf` and `scanf` are not part of the C programming language. For example, the compiler cannot find a spelling error in `printf` or `scanf`. When the compiler compiles a `printf` statement, it merely provides space in the object program for a “call” to the library function. But the compiler does not know where the library functions are—the linker does. When the linker runs, it locates the library functions and inserts the proper calls to these library functions in the object program. Now the object program is complete and ready to be executed. For this reason, the linked program is called an **executable**. If the function name is misspelled, it is the linker which will spot the error, because it will not be able to match the name in the C program with the name of any known function in the libraries.

The `printf` function can print **Welcome to C!** several different ways. For example, the program of Fig. 2.3 produces the same output as the program of Fig. 2.1. This works because each `printf` resumes printing where the previous `printf` stopped printing. The first `printf` (line 8) prints **Welcome** followed by a space and the second `printf` (line 9) begins printing on the same line immediately following the space.

```
/* Fig. 2.3: fig02_03.c */
/* Printing on one line with two printf statements */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
  printf( "Welcome ");
  printf( "to C!\n" );
  return 0; /* indicate that program ended successfully */
} /* end function main */
```

Fig. 2.3 | Printing on one line with two printf statements. (Part 1 of 2.)
One `printf` can print several lines by using additional newline characters as in Fig. 2.4. Each time the `\n` (newline) escape sequence is encountered, output continues at the beginning of the next line.

### 2.3 Another Simple C Program: Adding Two Integers

Our next program uses the Standard Library function `scanf` to obtain two integers typed by a user at the keyboard, computes the sum of these values and prints the result using `printf`. The program and sample output are shown in Fig. 2.8. [In the input/output dialog of Fig. 2.8, we emphasize the numbers input by the user in bold.]

```c
/* Fig. 2.5: fig02_05.c
Addition program */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    int integer1; /* first number to be input by user */
    int integer2; /* second number to be input by user */
    int sum; /* variable in which sum will be stored */

    printf( "Enter first integer\n" ); /* prompt */
    scanf( "%d", &integer1 ); /* read an integer */

    printf( "Enter second integer\n" ); /* prompt */
    scanf( "%d", &integer2 ); /* read an integer */

    sum = integer1 + integer2; /* compute sum */
    printf( "Sum of integers = %d\n", sum ); /* result */

    return 0; /* indicate that program ended successfully */
} /* end function main */
```
The comment in lines 1–2 states the purpose of the program. As we stated earlier, every program begins execution with `main`. The left brace `{` (line 7) marks the beginning of the body of `main` and the corresponding right brace `}` (line 24) marks the end of `main`. Lines 8–10 are definitions. The names `integer1`, `integer2` and `sum` are the names of variables. A variable is a location in memory where a value can be stored for use by a program. These definitions specify that the variables `integer1`, `integer2` and `sum` are of type `int`, which means that these variables will hold `integer` values, i.e., whole numbers such as 7, –11, 0, 31914 and the like. All variables must be defined with a name and a data type immediately after the left brace that begins the body of `main` before they can be used in a program. There are other data types besides `int` in C. The preceding definitions could have been combined into a single definition statement as follows:

```
int integer1, integer2, sum;
```

but that would have made it difficult to describe the variables in corresponding comments as we did in lines 8–10.

A variable name in C is any valid identifier. An identifier is a series of characters consisting of letters, digits and underscores (\_) that does not begin with a digit. An identifier can be of any length, but only the first 31 characters are required to be recognized by C compilers according to the C standard. C is case sensitive—uppercase and lowercase letters are different in C, so `a1` and `A1` are different identifiers.

**Common Programming Error 2.4**

Using a capital letter where a lowercase letter should be used (for example, typing `Main` instead of `main`).

**Error-Prevention Tip 2.1**

Avoid starting identifiers with the underscore character (\_) to prevent conflicts with compiler-generated identifiers and standard library identifiers.
Definitions must be placed after the left brace of a function and before any executable statements. For example, in the program illustrated in Fig. 2.5, inserting the definitions after the first `printf` would cause a syntax error. A syntax error is caused when the compiler cannot recognize a statement. The compiler normally issues an error message to help you locate and fix the incorrect statement. Syntax errors are violations of the language. Syntax errors are also called compile errors, or compile-time errors.

Line 12

```c
printf("Enter first integer\n"); /* prompt */
```

prints the literal `Enter first integer` on the screen and positions the cursor to the beginning of the next line. This message is called a prompt because it tells the user to take a specific action.

The next statement

```c
scanf("%d", &integer1); /* read an integer */
```

uses `scanf` to obtain a value from the user. The `scanf` function reads from the standard input, which is usually the keyboard. This `scanf` has two arguments, "%d" and `&integer1`. The first argument, the format control string, indicates the type of data that should be input by the user. The %d conversion specifier indicates that the data should be an integer.
(the letter \( d \) stands for “decimal integer”). The \% in this context is treated by \( \text{scanf} \) (and \( \text{printf} \) as we’ll see) as a special character that begins a conversion specifier. The second argument of \( \text{scanf} \) begins with an ampersand (\&)—called the \textit{address operator} in C—followed by the variable name. The ampersand, when combined with the variable name, tells \( \text{scanf} \) the location (or address) in memory at which the variable \( \text{integer1} \) is stored. The computer then stores the value for \( \text{integer1} \) at that location. The use of ampersand (\&) is often confusing to novice programmers or to people who have programmed in other languages that do not require this notation. For now, just remember to precede each variable in every call to \( \text{scanf} \) with an ampersand. Some exceptions to this rule are discussed in Chapters 6 and 7. The use of the ampersand will become clear after we study pointers in Chapter 7.

### Good Programming Practice 2.9

Place a space after each comma (,) to make programs more readable.

When the computer executes the preceding \( \text{scanf} \), it waits for the user to enter a value for variable \( \text{integer1} \). The user responds by typing an integer, then pressing the \textit{Enter key} to send the number to the computer. The computer then assigns this number, or value, to the variable \( \text{integer1} \). Any subsequent references to \( \text{integer1} \) in this program will use this same value. Functions \( \text{printf} \) and \( \text{scanf} \) facilitate interaction between the user and the computer. Because this interaction resembles a dialogue, it is often called \textit{conversational computing} or \textit{interactive computing}.

Line 15

\[
\text{printf( "Enter second integer\n" ); /* prompt */}
\]

displays the message \textit{Enter second integer} on the screen, then positions the cursor to the beginning of the next line. This \textit{printf} also prompts the user to take action.

The statement

\[
\text{scanf( "%d", &integer2 ); /* read an integer */}
\]

obtains a value for variable \( \text{integer2} \) from the user. The \textit{assignment statement} in line 18

\[
\text{sum} = \text{integer1} + \text{integer2}; /* assign total to sum */
\]

calculates the sum of variables \( \text{integer1} \) and \( \text{integer2} \) and assigns the result to variable \( \text{sum} \) using the assignment operator \( = \). The statement is read as, “\textit{sum gets the value of} \textit{integer1 + integer2}.” Most calculations are performed in assignments. The \( = \) operator and the \( + \) operator are called \textit{binary operators} because each has two \textit{operands}. The \( + \) operator’s two operands are \( \text{integer1} \) and \( \text{integer2} \). The \( = \) operator’s two operands are \textit{sum} and the value of the expression \( \text{integer1} + \text{integer2} \).

### Good Programming Practice 2.10

Place spaces on either side of a binary operator. This makes the operator stand out and makes the program more readable.

### Common Programming Error 2.6

A calculation in an assignment statement must be on the right side of the \( = \) operator. It is a compilation error to place a calculation on the left side of an assignment operator.
calls function printf to print the literal Sum is followed by the numerical value of variable sum on the screen. This printf has two arguments, "Sum is %d\n" and sum. The first argument is the format control string. It contains some literal characters to be displayed, and it contains the conversion specifier %d indicating that an integer will be printed. The second argument specifies the value to be printed. Notice that the conversion specifier for an integer is the same in both printf and scanf. This is the case for most C data types.

Calculations can also be performed inside printf statements. We could have combined the previous two statements into the statement

```
printf( "Sum is %d\n", integer1 + integer2 );
```

passes the value 0 back to the operating-system environment in which the program is being executed. This value indicates to the operating system that the program executed successfully. For information on how to report a program failure, see the manuals for your particular operating-system environment. The right brace, }, at line 24 indicates that the end of function main has been reached.

---

**Common Programming Error 2.7**

Forgetting one or both of the double quotes surrounding the format control string in a printf or scanf.

**Common Programming Error 2.8**

Forgetting the % in a conversion specification in the format control string of a printf or scanf.

**Common Programming Error 2.9**

Placing an escape sequence such as \n outside the format control string of a printf or scanf.

**Common Programming Error 2.10**

Forgetting to include the expressions whose values are to be printed in a printf containing conversion specifiers.

**Common Programming Error 2.11**

Not providing a conversion specifier when one is needed in a printf format control string to print the value of an expression.

**Common Programming Error 2.12**

Placing inside the format control string the comma that is supposed to separate the format control string from the expressions to be printed.
On many systems, the preceding execution-time error causes a “segmentation fault” or “access violation.” Such an error occurs when a user’s program attempts to access a part of the computer’s memory to which it does not have access privileges. The precise cause of this error will be explained in Chapter 7.

2.4 Memory Concepts

Variable names such as integer1, integer2 and sum actually correspond to locations in the computer’s memory. Every variable has a name, a type and a value.

In the addition program of Fig. 2.5, when the statement (line 13)

```c
scanf( "%d", &integer1 ); /* read an integer */
```

is executed, the value typed by the user is placed into a memory location to which the name integer1 has been assigned. Suppose the user enters the number 45 as the value for integer1. The computer will place 45 into location integer1 as shown in Fig. 2.6.

![Fig. 2.6](image1)

Whenever a value is placed in a memory location, the value replaces the previous value in that location; thus, placing a new value into a memory location is said to be destructive.

Returning to our addition program again, when the statement (line 16)

```c
scanf( "%d", &integer2 ); /* read an integer */
```

exectues, suppose the user enters the value 72. This value is placed into location integer2, and memory appears as in Fig. 2.7. These locations are not necessarily adjacent in memory.

Once the program has obtained values for integer1 and integer2, it adds these values and places the sum into variable sum. The statement (line 18)

```c
sum = integer1 + integer2; /* assign total to sum */
```

that performs the addition also replaces whatever value was stored in sum. This occurs when the calculated sum of integer1 and integer2 is placed into location sum (destroying the value already in sum). After sum is calculated, memory appears as in Fig. 2.8. The values of
Chapter 2  Introduction to C Programming

2.5 Arithmetic in C

Most C programs perform arithmetic calculations. The C arithmetic operators are summarized in Fig. 2.9. Note the use of various special symbols not used in algebra. The asterisk (*) indicates multiplication and the percent sign (%) denotes the remainder operator, which is introduced below. In algebra, if we want to multiply a times b, we can simply place these single-letter variable names side by side as in ab. In C, however, if we were to do this, ab would be interpreted as a single, two-letter name (or identifier). Therefore, C (and other programming languages, in general) require that multiplication be explicitly denoted by using the * operator as in a * b.

<table>
<thead>
<tr>
<th>C operation</th>
<th>Arithmetic operator</th>
<th>Algebraic expression</th>
<th>C expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
<td>f + 7</td>
<td>f + 7</td>
</tr>
<tr>
<td>Subtraction</td>
<td>−</td>
<td>p - c</td>
<td>p - c</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>bm</td>
<td>b * m</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>x / y or x (\frac{y}{x}) or x + y</td>
<td>x / y</td>
</tr>
<tr>
<td>Remainder</td>
<td>%</td>
<td>r mod s</td>
<td>r % s</td>
</tr>
</tbody>
</table>

The arithmetic operators are all binary operators. For example, the expression 3 + 7 contains the binary operator + and the operands 3 and 7.

Integer division yields an integer result. For example, the expression 7 / 4 evaluates to 1 and the expression 17 / 5 evaluates to 3. C provides the remainder operator, %, which
yields the remainder after integer division. The remainder operator is an integer operator that can be used only with integer operands. The expression \( x \% y \) yields the remainder after \( x \) is divided by \( y \). Thus, \( 7 \% 4 \) yields 3 and \( 17 \% 5 \) yields 2. We’ll discuss many interesting applications of the remainder operator.

### Arithmetic Expressions in Straight-Line Form

Arithmetic expressions in C must be written in straight-line form to facilitate entering programs into the computer. Thus, expressions such as “\( a \) divided by \( b \)” must be written as \( a/b \) so that all operators and operands appear in a straight line. The algebraic notation

\[
\frac{a}{b}
\]

is generally not acceptable to compilers, although some special-purpose software packages do support more natural notation for complex mathematical expressions.

### Parentheses for Grouping Subexpressions

Parentheses are used in C expressions in the same manner as in algebraic expressions. For example, to multiply \( a \) times the quantity \( b + c \) we write \( a \times (b + c) \).

### Rules of Operator Precedence

C applies the operators in arithmetic expressions in a precise sequence determined by the following rules of operator precedence, which are generally the same as those in algebra:

1. Operators in expressions contained within pairs of parentheses are evaluated first. Thus, parentheses may be used to force the order of evaluation to occur in any sequence you desire. Parentheses are said to be at the “highest level of precedence.” In cases of nested, or embedded, parentheses, such as

\[
( ( a + b ) + c )
\]

the operators in the innermost pair of parentheses are applied first.

2. Multiplication, division and remainder operations are applied first. If an expression contains several multiplication, division and remainder operations, evaluation proceeds from left to right. Multiplication, division and remainder are said to be on the same level of precedence.

3. Addition and subtraction operations are evaluated next. If an expression contains several addition and subtraction operations, evaluation proceeds from left to right. Addition and subtraction also have the same level of precedence, which is lower than the precedence of the multiplication, division and remainder operations.

The rules of operator precedence specify the order C uses to evaluate expressions. When we say evaluation proceeds from left to right, we’re referring to the associativity of the operators. We’ll see that some operators associate from right to left. Figure 2.10 summarizes these rules of operator precedence.

---

**Common Programming Error 2.16**

An attempt to divide by zero is normally undefined on computer systems and generally results in a fatal error, i.e., an error that causes the program to terminate immediately without having successfully performed its job. Nonfatal errors allow programs to run to completion, often producing incorrect results.
Chapter 2  Introduction to C Programming

Sample Algebraic and C Expressions

Now let’s consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its C equivalent. The following example calculates the arithmetic mean (average) of five terms.

\[ m = \frac{a + b + c + d + e}{5} \]

Java:

\[ m = (a + b + c + d + e) / 5; \]

The parentheses are required to group the additions because division has higher precedence than addition. The entire quantity \((a + b + c + d + e)\) should be divided by 5. If the parentheses are erroneously omitted, we obtain \(a + b + c + d + e / 5\) which evaluates incorrectly as

\[ a + b + c + d + \frac{e}{5} \]

The following example is the equation of a straight line:

Algebra: \( y = mx + b \)

C:

\[ y = m * x + b; \]

No parentheses are required. The multiplication is evaluated first because multiplication has a higher precedence than addition.

The following example contains remainder (%), multiplication, division, addition, subtraction and assignment operations:

Algebra: \[ z = pr\%q + w/x - y \]

C:

\[ z = p * r \% q + w / x - y; \]

1. We use simple examples to explain the order of evaluation of expressions. Subtle issues occur in more complex expressions that you’ll encounter later in the book. We’ll discuss these issues as they arise.
The circled numbers indicate the order in which C evaluates the operators. The multiplication, remainder and division are evaluated first in left-to-right order (i.e., they associate from left to right) since they have higher precedence than addition and subtraction. The addition and subtraction are evaluated next. They’re also evaluated left to right.

Not all expressions with several pairs of parentheses contain nested parentheses. For example, the following expression does not contain nested parentheses—instead, the parentheses are said to be “on the same level.”

\[ a \times (b + c) + c \times (d + e) \]

**Evaluation of a Second-Degree Polynomial**

To develop a better understanding of the rules of operator precedence, let’s see how C evaluates a second-degree polynomial.

\[
y = a \times x \times x + b \times x + c;
\]

The circled numbers under the statement indicate the order in which C performs the operations. There is no arithmetic operator for exponentiation in C, so we have represented \(x^2\) as \(x \times x\). The C Standard Library includes the `pow` (“power”) function to perform exponentiation. Because of some subtle issues related to the data types required by `pow`, we defer a detailed explanation of `pow` until Chapter 4.

Suppose variables \(a\), \(b\), \(c\) and \(x\) in the preceding second-degree polynomial are initialized as follows: \(a = 2\), \(b = 3\), \(c = 7\) and \(x = 5\). Figure 2.11 illustrates the order in which the operators are applied.

**Fig. 2.11** | Order in which a second-degree polynomial is evaluated.
Chapter 2  Introduction to C Programming

As in algebra, it is acceptable to place unnecessary parentheses in an expression to make the expression clearer. These are called redundant parentheses. For example, the preceding statement could be parenthesized as follows:

\[ y = ( a \times x \times x ) + ( b \times x ) + c; \]

Good Programming Practice 2.11
Using redundant parentheses in complex arithmetic expressions can make the expressions clearer.

2.6 Decision Making: Equality and Relational Operators

Executable C statements either perform actions (such as calculations or input or output of data) or make decisions (we’ll soon see several examples of these). We might make a decision in a program, for example, to determine if a person’s grade on an exam is greater than or equal to 60 and if it is to print the message “Congratulations! You passed.” This section introduces a simple version of C’s if statement that allows a program to make a decision based on the truth or falsity of a statement of fact called a condition. If the condition is met (i.e., the condition is true) the statement in the body of the if statement is executed. If the condition is not met (i.e., the condition is false) the body statement is not executed. Whether the body statement is executed or not, after the if statement completes, execution proceeds with the next statement after the if statement.

Conditions in if statements are formed by using the equality operators and relational operators summarized in Fig. 2.12. The relational operators all have the same level of precedence and they associate left to right. The equality operators have a lower level of precedence than the relational operators and they also associate left to right. [Note: In C, a condition may actually be any expression that generates a zero (false) or nonzero (true) value. We’ll see many applications of this throughout the book.]

<table>
<thead>
<tr>
<th>Algebraic equality or relational operator</th>
<th>C equality or relational operator</th>
<th>Example of C condition</th>
<th>Meaning of C condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>==</td>
<td>x == y</td>
<td>x is equal to y</td>
</tr>
<tr>
<td>≠</td>
<td>!=</td>
<td>x != y</td>
<td>x is not equal to y</td>
</tr>
<tr>
<td>Relational operators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>x &gt; y</td>
<td>x is greater than y</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>x &lt; y</td>
<td>x is less than y</td>
</tr>
<tr>
<td>≥</td>
<td>&gt;=</td>
<td>x &gt;= y</td>
<td>x is greater than or equal to y</td>
</tr>
<tr>
<td>≤</td>
<td>&lt;=</td>
<td>x &lt;= y</td>
<td>x is less than or equal to y</td>
</tr>
</tbody>
</table>

Fig. 2.12  | Equality and relational operators.

Common Programming Error 2.17
A syntax error occurs if the two symbols in any of the operators ==, !=, >= and <= are separated by spaces.
To avoid this confusion, the equality operator should be read “double equals” and the assignment operator should be read “gets” or “is assigned the value of.” As we’ll soon see, confusing these operators may not necessarily cause an easy-to-recognize compilation error, but may cause extremely subtle logic errors.

Figure 2.13 uses six if statements to compare two numbers input by the user. If the condition in any of these if statements is true, the printf statement associated with that if executes. The program and three sample execution outputs are shown in the figure.

```c
/* Fig. 2.13: fig02_13.c */
/* Using if statements, relational operators, and equality operators */
#include <stdio.h>

int main(void) {
    int num1; /* first number to be read from user */
    int num2; /* second number to be read from user */
    printf( "Enter two integers, and I will tell you the relationships they satisfy: " );
    scanf( "%d%d", &num1, &num2 ); /* read two integers */
    if ( num1 == num2 ) {
        printf( "%d is equal to %d\n", num1, num2 );
    } /* end if */
    if ( num1 != num2 ) {
        printf( "%d is not equal to %d\n", num1, num2 );
    } /* end if */
    if ( num1 < num2 ) {
        printf( "%d is less than %d\n", num1, num2 );
    } /* end if */

Fig. 2.13 | Using if statements, relational operators, and equality operators. (Part 1 of 2.)
Enter two integers, and I will tell you the relationships they satisfy:
3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7

Enter two integers, and I will tell you the relationships they satisfy:
12 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12

Enter two integers, and I will tell you the relationships they satisfy:
7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7

Fig. 2.13 | Using if statements, relational operators, and equality operators. (Part 2 of 2.)

The program uses scanf (line 15) to input two numbers. Each conversion specifier has a corresponding argument in which a value will be stored. The first %d converts a value to be stored in variable num1, and the second %d converts a value to be stored in variable num2. Indenting the body of each if statement and placing blank lines above and below each if statement enhances program readability.

**Good Programming Practice 2.12**
Indent the statement(s) in the body of an if statement.

**Good Programming Practice 2.13**
Place a blank line before and after every if statement in a program for readability.
A left brace, {, begins the body of each if statement (e.g., line 17). A corresponding right brace, }, ends each if statement’s body (e.g., line 19). Any number of statements can be placed in the body of an if statement.²

The comment (lines 1–3) in Fig. 2.13 is split over three lines. In C programs, white space characters such as tabs, newlines and spaces are normally ignored. So, statements and comments may be split over several lines. It is not correct, however, to split identifiers.

Figure 2.14 lists the precedence of the operators introduced in this chapter. Operators are shown top to bottom in decreasing order of precedence. The equals sign is also an operator. All these operators, with the exception of the assignment operator =, associate from left to right. The assignment operator (=) associates from right to left.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>left to right</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>=</td>
<td>right to left</td>
</tr>
</tbody>
</table>

Fig. 2.14 | Precedence and associativity of the operators discussed so far.

². Using braces to delimit the body of an if statement is optional when the body contains only one statement. Many programmers consider it good practice to always use these braces. In Chapter 3, we’ll explain the issues.
Some of the words we have used in the C programs in this chapter—in particular int, return and if—are **keywords** or reserved words of the language. Figure 2.15 contains the C keywords. These words have special meaning to the C compiler, so you must be careful not to use these as identifiers such as variable names. In this book, we discuss all these keywords.

<table>
<thead>
<tr>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
</tr>
<tr>
<td>break</td>
</tr>
<tr>
<td>case</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>const</td>
</tr>
<tr>
<td>continue</td>
</tr>
<tr>
<td>default</td>
</tr>
<tr>
<td>do</td>
</tr>
</tbody>
</table>

*Keywords added in C99*

_Bool _Complex _Imaginary inline restrict

**Fig. 2.15** | C’s keywords.

In this chapter, we have introduced many important features of the C programming language, including printing data on the screen, inputting data from the user, performing calculations and making decisions. In the next chapter, we build upon these techniques as we introduce structured programming. You’ll become more familiar with indentation techniques. We’ll study how to specify the order in which statements are executed—this is called **flow of control.**

**Summary**

**Section 2.1 Introduction**

- The C language facilitates a structured and disciplined approach to computer program design.

**Section 2.2 A Simple C Program: Printing a Line of Text**

- Comments begin with /* and end with */. Comments document programs and improve program readability. C99 also supports C++’s single-line comments that begin with //.
- Comments do not cause the computer to perform any action when the program is run. They’re ignored by the C compiler and do not cause any machine-language object code to be generated.
- Lines beginning with # are processed by the preprocessor before the program is compiled. The #include directive tells the preprocessor to include the contents of another file (typically a header file such as <stdio.h>).
- The <stdio.h> header contains information used by the compiler when compiling calls to standard input/output library functions such as printf.
• The function `main` is a part of every C program. The parentheses after `main` indicate that `main` is a program building block called a function. C programs contain one or more functions, one of which must be `main`. Every program in C begins executing at the function `main`.

• Functions can return information. The keyword `int` to the left of `main` indicates that `main` “returns” an integer (whole number) value.

• Functions can receive information when they’re called upon to execute. The `void` in parentheses after `main` indicates that `main` does not receive any information.

• A left brace, `{`, begins the body of every function. A corresponding right brace, `}`, ends each function. This pair of braces and the portion of the program between the braces is called a block.

• The `printf` function instructs the computer to display information on the screen.

• A string is sometimes called a character string, a message or a literal.

• Every statement must end with a semicolon (also known as the statement terminator).

• The characters `\n` do not display characters on the screen. The backslash (`\`) is called an escape character. When encountering a backslash in a string, the compiler looks ahead at the next character and combines it with the backslash to form an escape sequence. The escape sequence `\n` means newline.

• When a newline appears in the string output by a `printf`, the newline causes the cursor to position to the beginning of the next line on the screen.

• The double backslash (`\\`) escape sequence can be used to place a single backslash in a string.

• The escape sequence `\"` represents a literal double-quote character.

• The keyword `return` is one of several means to exit a function. When the `return` statement is used at the end of `main`, the value 0 indicates that the program has terminated successfully.

Section 2.3 Another Simple C Program: Adding Two Integers

• A variable is a location in memory where a value can be stored for use by a program.

• Variables of type `int` hold integer values, i.e., whole numbers such as 7, –11, 0, 31914.

• All variables must be defined with a name and a data type immediately after the left brace that begins the body of `main` before they can be used in a program.

• A variable name in C is any valid identifier. An identifier is a series of characters consisting of letters, digits and underscores (\_) that does not begin with a digit. An identifier can be any length, but only the first 31 characters are required to be recognized by C compilers according to the C standard.

• C is case sensitive—uppercase and lowercase letters are different in C.

• Definitions must be placed after the left brace of a function and before any executable statements.

• A syntax error is caused when the compiler cannot recognize a statement. The compiler normally issues an error message to help you locate and fix the incorrect statement. Syntax errors are violations of the language. Syntax errors are also called compile errors, or compile-time errors.

• Standard Library function `scanf` can be used to obtain input from the standard input, which is usually the keyboard.

• The `scanf` format control string indicates the type(s) of data that should be input.

• The `%d` conversion specifier indicates that the data should be an integer (the letter d stands for “decimal integer”). The % in this context is treated by `scanf` (and `printf`) as a special character that begins a conversion specifier.

• The other arguments of `scanf` begin with an ampersand (&)—called the address operator in C—followed by a variable name. The ampersand, when combined with a variable name, tells `scanf`
Chapter 2  Introduction to C Programming

the location in memory at which the variable is located. The computer then stores the value for
the variable at that location.

• Most calculations are performed in assignment statements.

• The = operator and the + operator are binary operators—each has two operands.

• Function printf also can use a format control string as its first argument. This string contains
some literal characters to be displayed and the conversion specifiers that indicate place holders
for data to output.

Section 2.4 Memory Concepts

• Variable names correspond to locations in the computer’s memory. Every variable has a name, a
type and a value.

• Whenever a value is placed in a memory location, the value replaces the previous value in that
location; thus, placing a new value into a memory location is said to be destructive.

• When a value is read out of a memory location, the process is said to be nondestructive.

Section 2.5 Arithmetic in C

• In algebra, if we want to multiply a times b, we can simply place these single-letter variable names
side by side as in ab. In C, however, if we were to do this, ab would be interpreted as a single,
two-letter name (or identifier). Therefore, C (like other programming languages, in general) re-
quires that multiplication be explicitly denoted by using the * operator, as in a * b.

• The arithmetic operators are all binary operators.

• Integer division yields an integer result. For example, the expression 7 / 4 evaluates to 1 and the
expression 17 / 5 evaluates to 3.

• C provides the remainder operator, %, which yields the remainder after integer division. The re-
mainder operator is an integer operator that can be used only with integer operands. The expres-
sion x % y yields the remainder after x is divided by y. Thus, 7 % 4 yields 3 and 17 % 5 yields 2.

• An attempt to divide by zero is normally undefined on computer systems and generally results in
a fatal error that causes the program to terminate immediately. Nonfatal errors allow programs
to run to completion, often producing incorrect results.

• Arithmetic expressions in C must be written in straight-line form to facilitate entering programs
into the computer. Thus, expressions such as “a divided by b” must be written as a/b so that all
operators and operands appear in a straight line.

• Parentheses are used to group terms in C expressions in much the same manner as in algebraic
expressions.

• C evaluates arithmetic expressions in a precise sequence determined by the following rules of op-
erator precedence, which are generally the same as those followed in algebra.

• Multiplication, division and remainder operations are applied first. If an expression contains sev-
eral multiplication, division and remainder operations, evaluation proceeds from left to right.
Multiplication, division and remainder are said to be on the same level of precedence.

• Addition and subtraction operations are evaluated next. If an expression contains several addition
and subtraction operations, evaluation proceeds from left to right. Addition and subtraction also
have the same level of precedence, which is lower than the precedence of the multiplication, di-
vision and remainder operators.

• The rules of operator precedence specify the order C uses to evaluate expressions. When we say
evaluation proceeds from left to right, we’re referring to the associativity of the operators. Some
operators associate from right to left.
Section 2.6 Decision Making: Equality and Relational Operators

- Executable C statements either perform actions or make decisions.
- C’s if statement allows a program to make a decision based on the truth or falsity of a statement of fact called a condition. If the condition is met (i.e., the condition is true) the statement in the body of the if statement executes. If the condition is not met (i.e., the condition is false) the body statement does not execute. Whether the body statement is executed or not, after the if statement completes, execution proceeds with the next statement after the if statement.
- Conditions in if statements are formed by using the equality operators and relational operators.
- The relational operators all have the same level of precedence and associate left to right. The equality operators have a lower level of precedence than the relational operators and they also associate left to right.
- To avoid confusing assignment (=) and equality (==), the assignment operator should be read “gets” and the equality operator should be read “double equals.”
- In C programs, white-space characters such as tabs, newlines and spaces are normally ignored. So, statements and comments may be split over several lines. It is not correct to split identifiers.
- Some of the words in C programs—such as int, return and if—are keywords or reserved words of the language. These words have special meaning to the C compiler, so you cannot use them as identifiers such as variable names.

Terminology

* multiplication operator 34
% remainder operator 34
%d conversion specifier 30
action 26
action/decision model 27
address operator (&) 31
argument 26
arithmetic operators 34
assignment statement 31
associativity 35
body 25
C preprocessor 25
case sensitive 29
character string 26
comment (/\* \*/ ) 24
compile error 30
compile-time error 30
condition 38
conversational computing 31
decision 38
definition 29
destructive 33
document a program 24
embedded parentheses 35
Enter key 31
equality operator 38
escape character 26
escape sequence 26
executable 27
exit a function 26
false 38
flow of control 42
format control string 30
function 25
identifier 29
if statement 38
integer 29
integer division 34
interactive computing 31
keyword 42
literal 26
message 26
nested parentheses 35
newline (\n) 26
nondestructive 34
operand 31
percent sign (%) 34
prompt 30
redundant parentheses 38
relational operator 38
right brace (}) 25
rules of operator precedence 35
scanf function 30
single-line comment (// ) 25
standard input/output header 25
statement 26
Self-Review Exercises

2.1 Fill in the blanks in each of the following.

a) Every C program begins execution at the function ________.

b) The ________ begins the body of every function and the ________ ends the body of every function.

c) Every statement ends with a(n) ________.

d) The ________ standard library function displays information on the screen.

e) The escape sequence \n represents the ________ character, which causes the cursor to position to the beginning of the next line on the screen.

f) The ________ Standard Library function is used to obtain data from the keyboard.

g) The conversion specifier ________ is used in a scanf format control string to indicate that an integer will be input and in a printf format control string to indicate that an integer will be output.

h) Whenever a new value is placed in a memory location, that value overrides the previous value in that location. This process is said to be ________.

i) When a value is read out of a memory location, the value in that location is preserved; this process is said to be ________.

j) The ________ statement is used to make decisions.

2.2 State whether each of the following is true or false. If false, explain why.

a) Function printf always begins printing at the beginning of a new line.

b) Comments cause the computer to print the text enclosed between /* and */ on the screen when the program is executed.

c) The escape sequence \n when used in a printf format control string causes the cursor to position to the beginning of the next line on the screen.

d) All variables must be defined before they’re used.

e) All variables must be given a type when they’re defined.

f) C considers the variables number and NuMbEr to be identical.

g) Definitions can appear anywhere in the body of a function.

h) All arguments following the format control string in a printf function must be preceded by an ampersand (&).

i) The remainder operator (%) can be used only with integer operands.

j) The arithmetic operators *, /, %, + and - all have the same level of precedence.

k) The following variable names are identical on all Standard C systems.

\texttt{thisisasuperduperlongname1234567}
\texttt{thisisasuperduperlongname1234568}

l) A program that prints three lines of output must contain three printf statements.

2.3 Write a single C statement to accomplish each of the following:

a) Define the variables \texttt{c}, \texttt{thisVariable}, \texttt{q76354} and \texttt{number} to be of type \texttt{int}.

b) Prompt the user to enter an integer. End your prompting message with a colon (:) followed by a space and leave the cursor positioned after the space.

c) Read an integer from the keyboard and store the value entered in integer variable \texttt{a}.

d) If \texttt{number} is not equal to \texttt{7}, print “The variable number is not equal to \texttt{7}.”
2.4 Write a statement (or comment) to accomplish each of the following:
   a) State that a program will calculate the product of three integers.
   b) Define the variables x, y, z and result to be of type int.
   c) Prompt the user to enter three integers.
   d) Read three integers from the keyboard and store them in the variables x, y and z.
   e) Compute the product of the three integers contained in variables x, y and z, and assign the result to the variable result.
   f) Print "The product is" followed by the value of the integer variable result.

2.5 Using the statements you wrote in Exercise 2.4, write a complete program that calculates the product of three integers.

2.6 Identify and correct the errors in each of the following statements:
   a) printf("The value is %d\n", &number);
   b) scanf("%d%d", &number1, number2);
   c) if ( c < 7 ){
       printf("C is less than 7\n");
   }
   d) if ( c => 7 ) {
       printf("C is equal to or less than 7\n");
   }

Answers to Self-Review Exercises

2.1 a) main. b) left brace ({), right brace (}), c) semicolon. d) printf. e) newline. f) scanf.
g) %d. h) destructive. i) nondestructive. j) if.

2.2 a) False. Function printf always begins printing where the cursor is positioned, and this may be anywhere on a line of the screen.
   b) False. Comments do not cause any action to be performed when the program is executed. They’re used to document programs and improve their readability.
   c) True.
   d) True.
   e) True.
   f) False. C is case sensitive, so these variables are unique.
   g) False. The definitions must appear after the left brace of the body of a function and before any executable statements.
   h) False. Arguments in a printf function ordinarily should not be preceded by an ampersand. Arguments following the format control string in a scanf function ordinarily should be preceded by an ampersand. We’ll discuss exceptions to these rules in Chapter 6 and Chapter 7.
   i) True.
   j) False. The operators *, / and % are on the same level of precedence, and the operators + and - are on a lower level of precedence.
   k) False. Some systems may distinguish between identifiers longer than 31 characters.
   l) False. A printf statement with multiple \n escape sequences can print several lines.

2.3 a) int c, thisVariable, q76354, number;
   b) printf("Enter an integer: ");
c) `scanf( "%d", &a );`

d) `if ( number != 7 )`
```
{
    printf( "The variable number is not equal to 7.\n" );
}
```
e) `printf( "This is a C program.\n" );`
f) `printf( "This is a C\nprogram.\n" );`
g) `printf( "This\nis\na\nC\nprogram.\n" );`
h) `printf( "This\tis\ta\tC\tprogram.\n" );`

2.4 a) `/* Calculate the product of three integers */`
b) `int x, y, z, result;`
c) `printf( "Enter three integers: " );`
d) `scanf( "%d%d%d", &x, &y, &z );`
e) `result = x * y * z;`
f) `printf( "The product is %d\n", result );`

2.5 See below.

2.6 a) Error: `&number`. Correction: Eliminate the &. We discuss exceptions to this later.
b) Error: `number2` does not have an ampersand. Correction: `number2` should be `&number2`. Later in the text we discuss exceptions to this.
c) Error: Semicolon after the right parenthesis of the condition in the `if` statement. Correction: Remove the semicolon after the right parenthesis. [Note: The result of this error is that the `printf` statement will be executed whether or not the condition in the `if` statement is true. The semicolon after the right parenthesis is considered an empty statement—a statement that does nothing.]
d) Error: The relational operator `=>` should be changed to `>=` (greater than or equal to).

Exercises

2.7 Identify and correct the errors in each of the following statements. (Note: There may be more than one error per statement.)
a) `scanf( "d", value );`
b) `printf( "The product of %d and %d is %d\n", x, y );`
c) `firstNumber + secondNumber = sumOfNumbers`
d) `if ( number => largest )`
```
    largest = number;
```
e) `*/ Program to determine the largest of three integers */`
f) `Scanf( "%d", anInteger );`
Exercises

49

g) printf( "Remainder of %d divided by %d is\n", x, y, x % y );

h) if ( x = y );
   printf( %d is equal to %d\n", x, y );

i) printf( "The sum is %d\n", x + y );

j) printf( "The value you entered is: %d\n", &value );

2.8 Fill in the blanks in each of the following:
a) _______ are used to document a program and improve its readability.
b) The function used to display information on the screen is ________.
c) A C statement that makes a decision is ________.
d) Calculations are normally performed by ________ statements.
e) The ________ function inputs values from the keyboard.

2.9 Write a single C statement or line that accomplishes each of the following:
a) Print the message “Enter two numbers.”
b) Assign the product of variables b and c to variable a.
c) State that a program performs a sample payroll calculation (i.e., use text that helps to document a program).
d) Input three integer values from the keyboard and place these values in integer variables a, b and c.

2.10 State which of the following are true and which are false. If false, explain your answer.
a) C operators are evaluated from left to right.
b) The following are all valid variable names: _under_bar_, m928134, t5, j7, her_sales, his_account_total, a, b, c, z, z2.
c) The statement printf("a = 5;""); is a typical example of an assignment statement.
d) A valid arithmetic expression containing no parentheses is evaluated from left to right.
e) The following are all invalid variable names: 3g, 87, 67h2, h22, 2h.

2.11 Fill in the blanks in each of the following:
a) What arithmetic operations are on the same level of precedence as multiplication?

b) When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression?

c) A location in the computer’s memory that may contain different values at various times throughout the execution of a program is called a ________.

2.12 What, if anything, prints when each of the following statements is performed? If nothing prints, then answer “Nothing.” Assume x = 2 and y = 3.
a) printf( "%d", x );
b) printf( "%d", x + x );
c) printf( "x=" );
d) printf( "x=%d", x );
e) printf( "%d = %d", x + y, y + x );
f) z = x + y;
g) scanf( "%d%d", &x, &y );
h) /* printf( "x + y = %d", x + y ); */
i) printf( "\n" );

2.13 Which, if any, of the following C statements contain variables whose values are replaced?
a) scanf( "%d%d%d%d%d", &b, &c, &d, &e, &f );
b) p = i + j + k + 7;
c) printf( "Values are replaced" );
d) printf( "a = 5" );
Given the equation \( y = ax^3 + 7 \), which of the following, if any, are correct C statements for this equation?

a) \( y = a * x * x * x + 7; \)

b) \( y = a * x * x * (x + 7); \)

c) \( y = (a * x) * x * (x + 7); \)

d) \( y = (a * x) * x * x + 7; \)

e) \( y = a * (x * x * x) + 7; \)

f) \( y = a * x * (x * x + 7); \)

State the order of evaluation of the operators in each of the following C statements and show the value of \( x \) after each statement is performed.

a) \( x = 7 + 3 * 6 / 2 - 1; \)

b) \( x = 2 \% 2 + 2 * 2 - 2 / 2; \)

c) \( x = (3 * 9 * (3 + (9 * 3 / (3)))) \)

Write a program that asks the user to enter two numbers, obtains them from the user and prints their sum, product, difference, quotient and remainder.

Write a program that prints the numbers 1 to 4 on the same line. Write the program using the following methods.

a) Using one \printf{} statement with no conversion specifiers.

b) Using one \printf{} statement with four conversion specifiers.

c) Using four \printf{} statements.

Write a program that asks the user to enter two integers, obtains the numbers from the user, then prints the larger number followed by the words “is larger.” If the numbers are equal, print the message “These numbers are equal.” Use only the single-selection form of the if statement you learned in this chapter.

Write a program that inputs three different integers from the keyboard, then prints the sum, the average, the product, the smallest and the largest of these numbers. Use only the single-selection form of the if statement you learned in this chapter. The screen dialogue should appear as follows:

- Input three different integers: 13 27 14
- Sum is 54
- Average is 18
- Product is 4914
- Smallest is 13
- Largest is 27

Write a program that reads in the radius of a circle and prints the circle’s diameter, circumference and area. Use the constant value 3.14159 for \( \pi \). Perform each of these calculations inside the \printf{} statement(s) and use the conversion specifier \%f. [Note: In this chapter, we have discussed only integer constants and variables. In Chapter 3 we’ll discuss floating-point numbers, i.e., values that can have decimal points.]

Write a program that prints the following shapes with asterisks.
2.22 What does the following code print?
```
printf("\n\n***\n****\n*****\n\n");
```

2.23 (Largest and Smallest Integers) Write a program that reads in five integers and then determines and prints the largest and the smallest integers in the group. Use only the programming techniques you have learned in this chapter.

2.24 (Odd or Even) Write a program that reads an integer and determines and prints whether it is odd or even. [Hint: Use the remainder operator. An even number is a multiple of two. Any multiple of two leaves a remainder of zero when divided by 2.]

2.25 Print your initials in block letters down the page. Construct each block letter out of the letter it represents as shown below.

```
PPPPPPPPP
 P P
 P P
 P P
 JJ
 J
 J
 JJJJJJJJ
DDDDDDDDD
 D D
 D D
 D D
 DDDD
```

2.26 (Multiples) Write a program that reads in two integers and determines and prints if the first is a multiple of the second. [Hint: Use the remainder operator.]

2.27 (Checkerboard Pattern of Asterisks) Display the following checkerboard pattern with eight `printf` statements and then display the same pattern with as few `printf` statements as possible.

```
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
```

2.28 Distinguish between the terms fatal error and nonfatal error. Why might you prefer to experience a fatal error rather than a nonfatal error?

2.29 (Integer Value of a Character) Here’s a peek ahead. In this chapter you learned about integers and the type `int`. C can also represent uppercase letters, lowercase letters and a considerable variety of special symbols. C uses small integers internally to represent each different character. The set of characters a computer uses together with the corresponding integer representations for those characters is called that computer’s character set. You can print the integer equivalent of uppercase A, for example, by executing the statement
```
printf("%d", 'A');
```

Write a C program that prints the integer equivalents of some uppercase letters, lowercase letters, digits and special symbols. As a minimum, determine the integer equivalents of the following: A B C a b c 0 1 2 $ * + / and the blank character.
2.30  **(Separating Digits in an Integer)** Write a program that inputs one five-digit number, separates the number into its individual digits and prints the digits separated from one another by three spaces each. [*Hint: Use combinations of integer division and the remainder operation.*] For example, if the user types in 42139, the program should print

```
4 2 1 3 9
```

2.31  **(Table of Squares and Cubes)** Using only the techniques you learned in this chapter, write a program that calculates the squares and cubes of the numbers from 0 to 10 and uses tabs to print the following table of values:

<table>
<thead>
<tr>
<th>number</th>
<th>square</th>
<th>cube</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>125</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>216</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
<td>343</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
<td>729</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
</tbody>
</table>

**Making a Difference**

2.32  **(Body Mass Index Calculator)** We introduced the body mass index (BMI) calculator in Exercise 1.11. The formulas for calculating BMI are

\[
BMI = \frac{weightInPounds \times 703}{heightInInches \times heightInInches}
\]

or

\[
BMI = \frac{weightInKilograms}{heightInMeters \times heightInMeters}
\]

Create a BMI calculator application that reads the user’s weight in pounds and height in inches (or, if you prefer, the user’s weight in kilograms and height in meters), then calculates and displays the user’s body mass index. Also, the application should display the following information from the Department of Health and Human Services/National Institutes of Health so the user can evaluate his/her BMI:

**BMI VALUES**

- Underweight: less than 18.5
- Normal: between 18.5 and 24.9
- Overweight: between 25 and 29.9
- Obese: 30 or greater

*[Note: In this chapter, you learned to use the int type to represent whole numbers. The BMI calculations when done with int values will both produce whole-number results. In Chapter 4 you’ll learn to use the double type to represent numbers with decimal points. When the BMI calculations are performed with doubles, they’ll both produce numbers with decimal points—these are called “floating-point” numbers.]*
2.33  (Car-Pool Savings Calculator) Research several car-pooling websites. Create an application that calculates your daily driving cost, so that you can estimate how much money could be saved by car pooling, which also has other advantages such as reducing carbon emissions and reducing traffic congestion. The application should input the following information and display the user’s cost per day of driving to work:

   a) Total miles driven per day.
   b) Cost per gallon of gasoline.
   c) Average miles per gallon.
   d) Parking fees per day.
   e) Tolls per day.
Let’s all move one place on.
—Lewis Carroll

The wheel is come full circle.
—William Shakespeare

How many apples fell on Newton’s head before he took the hint!
—Robert Frost

All the evolution we know of proceeds from the vague to the definite.
—Charles Sanders Peirce

Objectives
In this chapter, you’ll learn:

- Basic problem-solving techniques.
- To develop algorithms through the process of top-down, stepwise refinement.
- To use the if selection statement and the if...else selection statement to select actions.
- To use the while repetition statement to execute statements in a program repeatedly.
- Counter-controlled repetition and sentinel-controlled repetition.
- Structured programming.
- The increment, decrement and assignment operators.
3.1 Introduction

Before writing a program to solve a particular problem, it’s essential to have a thorough understanding of the problem and a carefully planned approach to solving the problem. The next two chapters discuss techniques that facilitate the development of structured computer programs. In Section 4.12, we present a summary of structured programming that ties together the techniques developed here and in Chapter 4.

3.2 Algorithms

The solution to any computing problem involves executing a series of actions in a specific order. A procedure for solving a problem in terms of

1. the actions to be executed, and
2. the order in which these actions are to be executed

is called an algorithm. The following example demonstrates that correctly specifying the order in which the actions are to be executed is important.

Consider the “rise-and-shine algorithm” followed by one junior executive for getting out of bed and going to work: Consider the “rise-and-shine algorithm” followed by one junior executive for getting out of bed and going to work: (1) Get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast, (6) carpool to work. This routine gets the executive to work well prepared to make critical decisions. Suppose that the same steps are performed in a slightly different order: (1) Get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work. In this case, our junior executive shows up for work soaking wet. Specifying the order in which statements are to be executed in a computer program is called program control. In this and the next chapter, we investigate the program control capabilities of C.

3.3 Pseudocode

Pseudocode is an artificial and informal language that helps you develop algorithms. The pseudocode we present here is particularly useful for developing algorithms that will be converted to structured C programs. Pseudocode is similar to everyday English; it’s convenient and user friendly although it’s not an actual computer programming language.
Chapter 3  Structured Program Development in C

Pseudocode programs are not executed on computers. Rather, they merely help you “think out” a program before attempting to write it in a programming language such as C. In this chapter, we give several examples of how pseudocode may be used effectively in developing structured C programs.

Pseudocode consists purely of characters, so you may conveniently type pseudocode programs into a computer using an editor program. The computer can display or print a fresh copy of a pseudocode program on demand. A carefully prepared pseudocode program may be converted easily to a corresponding C program. This is done in many cases simply by replacing pseudocode statements with their C equivalents.

Pseudocode consists only of action statements—those that are executed when the program has been converted from pseudocode to C and is run in C. Definitions are not executable statements. They’re messages to the compiler. For example, the definition

```
int i;
```

simply tells the compiler the type of variable i and instructs the compiler to reserve space in memory for the variable. But this definition does not cause any action—such as input, output, or a calculation—to occur when the program is executed. Some programmers choose to list each variable and briefly mention the purpose of each at the beginning of a pseudocode program.

3.4 Control Structures

Normally, statements in a program are executed one after the other in the order in which they’re written. This is called sequential execution. Various C statements we’ll soon discuss enable you to specify that the next statement to be executed may be other than the next one in sequence. This is called transfer of control.

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of a great deal of difficulty experienced by software development groups. The finger of blame was pointed at the goto statement that allows programmers to specify a transfer of control to one of many possible destinations in a program. The notion of so-called structured programming became almost synonymous with “goto elimination.”

The research of Bohm and Jacopini\(^1\) had demonstrated that programs could be written without any goto statements. The challenge of the era was for programmers to shift their styles to “goto-less programming.” It was not until well into the 1970s that the programming profession started taking structured programming seriously. The results were impressive, as software development groups reported reduced development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. Programs produced with structured techniques were clearer, easier to debug and modify and more likely to be bug free in the first place.

Bohm and Jacopini’s work demonstrated that all programs could be written in terms of only three control structures, namely the sequence structure, the selection structure and the repetition structure. The sequence structure is built into C. Unless directed otherwise, the computer executes C statements one after the other in the order in which they’re written. The flowchart segment of Fig. 3.1 illustrates C’s sequence structure.

---

A flowchart is a graphical representation of an algorithm or of a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols such as rectangles, diamonds, ovals, and small circles; these symbols are connected by arrows called flowlines.

Like pseudocode, flowcharts are useful for developing and representing algorithms, although pseudocode is preferred by most programmers. Flowcharts clearly show how control structures operate; that is all we use them for in this text.

Consider the flowchart for the sequence structure in Fig. 3.1. We use the rectangle symbol, also called the action symbol, to indicate any type of action including a calculation or an input/output operation. The flowlines in the figure indicate the order in which the actions are performed—first, grade is added to total, then 1 is added to counter. C allows us to have as many actions as we want in a sequence structure. As we’ll soon see, anywhere a single action may be placed, we may place several actions in sequence.

Figure 3.1 Flowcharting C’s sequence structure.

When drawing a flowchart that represents a complete algorithm, an oval symbol containing the word “Begin” is the first symbol used in the flowchart; an oval symbol containing the word “End” is the last symbol used. When drawing only a portion of an algorithm as in Fig. 3.1, the oval symbols are omitted in favor of using small circle symbols, also called connector symbols.

Perhaps the most important flowcharting symbol is the diamond symbol, also called the decision symbol, which indicates that a decision is to be made. We’ll discuss the diamond symbol in the next section.

C provides three types of selection structures in the form of statements. The if selection statement (Section 3.5) either performs (selects) an action if a condition is true or skips the action if the condition is false. The if…else selection statement (Section 3.6) performs an action if a condition is true and performs a different action if the condition is false. The switch selection statement (discussed in Chapter 4) performs one of many different actions depending on the value of an expression. The if statement is called a single-selection statement because it selects or ignores a single action. The if…else statement is called a double-selection statement because it selects between two different actions. The switch statement is called a multiple-selection statement because it selects among many different actions.

C provides three types of repetition structures in the form of statements, namely while (Section 3.7), do...while, and for (both discussed in Chapter 4).
That is all there is. C has only seven control statements: sequence, three types of selection and three types of repetition. Each C program is formed by combining as many of each type of control statement as is appropriate for the algorithm the program implements. As with the sequence structure of Fig. 3.1, we’ll see that the flowchart representation of each control statement has two small circle symbols, one at the entry point to the control statement and one at the exit point. These single-entry/single-exit control statements make it easy to build programs. The control-statement flowchart segments can be attached to one another by connecting the exit point of one control statement to the entry point of the next. This is much like the way in which a child stacks building blocks, so we call this control-statement stacking. We’ll learn that there is only one other way control statements may be connected—a method called control-statement nesting. Thus, any C program we’ll ever need to build can be constructed from only seven different types of control statements combined in only two ways. This is the essence of simplicity.

3.5 The if Selection Statement

Selection structures are used to choose among alternative courses of action. For example, suppose the passing grade on an exam is 60. The pseudocode statement

| If student’s grade is greater than or equal to 60  
| Print “Passed” |

determines if the condition “student’s grade is greater than or equal to 60” is true or false. If the condition is true, then “Passed” is printed, and the next pseudocode statement in order is “performed” (remember that pseudocode is not a real programming language). If the condition is false, the printing is ignored, and the next pseudocode statement in order is performed. The second line of this selection structure is indented. Such indentation is optional, but it’s highly recommended as it helps emphasize the inherent structure of structured programs. We’ll apply indentation conventions carefully throughout this text. The C compiler ignores white-space characters like blanks, tabs and newlines used for indentation and vertical spacing.

The preceding pseudocode If statement may be written in C as

```c
if ( grade >= 60 ) {
    printf( "Passed\n" );
} /* end if */
```

Notice that the C code corresponds closely to the pseudocode. This is one of the properties of pseudocode that makes it such a useful program development tool.
The flowchart of Fig. 3.2 illustrates the single-selection if statement. This flowchart contains what is perhaps the most important flowcharting symbol—the diamond symbol, also called the decision symbol, which indicates that a decision is to be made. The decision symbol contains an expression, such as a condition, that can be either true or false. The decision symbol has two flowlines emerging from it. One indicates the direction to take when the expression in the symbol is true; the other indicates the direction to take when the expression is false. Decisions can be based on conditions containing relational or equality operators. In fact, a decision can be based on any expression—if the expression evaluates to zero, it’s treated as false, and if it evaluates to nonzero, it’s treated as true.

The if selection statement performs an indicated action only when the condition is true; otherwise the action is skipped. The if…else selection statement allows you to specify that different actions are to be performed when the condition is true than when the condition is false. For example, the pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
else
    Print "Failed"
```

prints Passed if the student’s grade is greater than or equal to 60 and prints Failed if the student’s grade is less than 60. In either case, after printing occurs, the next pseudocode
The body of the else is also indented. Whatever indentation convention you choose should be carefully applied throughout your programs. It's difficult to read a program that does not obey uniform spacing conventions.

**Good Programming Practice 3.3**

*Indent both body statements of an if...else statement.*

**Good Programming Practice 3.4**

*If there are several levels of indentation, each level should be indented the same additional amount of space.*

The preceding pseudocode If...else statement may be written in C as

```c
if ( grade >= 60 ) {
    printf( "Passed\n" );
} /* end if */
else {
    printf( "Failed\n" );
} /* end else */
```

The flowchart of Fig. 3.3 nicely illustrates the flow of control in the if...else statement. Once again, note that (besides small circles and arrows) the only symbols in the flowchart are rectangles (for actions) and a diamond (for a decision). We continue to emphasize this action/decision model of computing. Imagine again a deep bin containing as many empty double-selection statements (represented as flowchart segments) as might be needed to build any C program. Your job, again, is to assemble these selection statements (by stacking and nesting) with any other control statements required by the algorithm, and to fill in the empty rectangles and empty diamonds with actions and decisions appropriate to the algorithm being implemented.

**Fig. 3.3** | Flowcharting the double-selection if...else statement.

C provides the **conditional operator** (?:) which is closely related to the if...else statement. The conditional operator is C's only ternary operator—it takes three operands. The operands together with the conditional operator form a **conditional expression**. The first operand is a condition. The second operand is the value for the entire conditional expression if the condition is true and the third operand is the value for the entire conditional expression if the condition is false. For example, the `printf` statement
contains a conditional expression that evaluates to the string literal "Passed" if the condition grade >= 60 is true and evaluates to the string literal "Failed" if the condition is false. The format control string of the printf contains the conversion specification %s for printing a character string. So the preceding printf statement performs in essentially the same way as the preceding if...else statement.

The second and third operands in a conditional expression can also be actions to be executed. For example, the conditional expression

```c
grade >= 60 ? printf( "Passed\n" ) : printf( "Failed\n" );
```

is read, “If grade is greater than or equal to 60 then printf("Passed\n"), otherwise printf( "Failed\n" ).” This, too, is comparable to the preceding if...else statement. We’ll see that conditional operators can be used in some situations where if...else statements cannot.

**Nested if...else statements** test for multiple cases by placing if...else statements inside if...else statements. For example, the following pseudocode statement will print A for exam grades greater than or equal to 90, B for grades greater than or equal to 80, C for grades greater than or equal to 70, D for grades greater than or equal to 60, and F for all other grades.

```plaintext
If student's grade is greater than or equal to 90
    Print "A"
else
    If student's grade is greater than or equal to 80
        Print "B"
    else
        If student's grade is greater than or equal to 70
            Print "C"
        else
            If student's grade is greater than or equal to 60
                Print "D"
            else
                Print "F"
```

This pseudocode may be written in C as

```c
if ( grade >= 90 )
    printf( "A\n" );
else
    if ( grade >= 80 )
        printf("B\n");
    else
        if ( grade >= 70 )
            printf("C\n");
        else
            if ( grade >= 60 )
                printf( "D\n" );
            else
                printf( "F\n" );
```
If the variable grade is greater than or equal to 90, the first four conditions will be true, but only the printf statement after the first test will be executed. After that printf is executed, the else part of the “outer” if...else statement is skipped. Many C programmers prefer to write the preceding if statement as

```c
if ( grade >= 90 )
  printf( "A\n" );
else if ( grade >= 80 )
  printf( "B\n" );
else if ( grade >= 70 )
  printf( "C\n" );
else if ( grade >= 60 )
  printf( "D\n" );
else
  printf( "F\n" );
```

As far as the C compiler is concerned, both forms are equivalent. The latter form is popular because it avoids the deep indentation of the code to the right. Such indentation often leaves little room on a line, forcing lines to be split and decreasing program readability.

The if selection statement expects only one statement in its body. To include several statements in the body of an if, enclose the set of statements in braces ({ and }). A set of statements contained within a pair of braces is called a compound statement or a block.

**Software Engineering Observation 3.1**

A compound statement can be placed anywhere in a program that a single statement can be placed.

The following example includes a compound statement in the else part of an if...else statement.

```c
if ( grade >= 60 ) {
  printf( "Passed.\n" );
} /* end if */
else {
  printf( "Failed.\n" );
  printf( "You must take this course again.\n" );
} /* end else */
```

In this case, if grade is less than 60, the program executes both printf statements in the body of the else and prints

```
Failed.
You must take this course again.
```

Notice the braces surrounding the two statements in the else clause. These braces are important. Without the braces, the statement

```c
printf( "You must take this course again.\n" );
```

would be outside the body of the else part of the if, and would execute regardless of whether the grade was less than 60.

**Common Programming Error 3.1**

Forgetting one or both of the braces that delimit a compound statement.
A syntax error is caught by the compiler. A logic error has its effect at execution time. A fatal logic error causes a program to fail and terminate prematurely. A nonfatal logic error allows a program to continue executing but to produce incorrect results.

Software Engineering Observation 3.2
Just as a compound statement can be placed anywhere a single statement can be placed, it's also possible to have no statement at all, i.e., the empty statement. The empty statement is represented by placing a semicolon (;) where a statement would normally be.

Common Programming Error 3.2
Placing a semicolon after the condition in an if statement as in if (grade >= 60); leads to a logic error in single-selection if statements and a syntax error in double-selection if statements.

Error-Prevention Tip 3.1
Typing the beginning and ending braces of compound statements before typing the individual statements within the braces helps avoid omitting one or both of the braces, preventing syntax errors and logic errors (where both braces are indeed required).

3.7 The while Repetition Statement
A repetition statement allows you to specify that an action is to be repeated while some condition remains true. The pseudocode statement

```
While there are more items on my shopping list
    Purchase next item and cross it off my list
```

describes the repetition that occurs during a shopping trip. The condition, “there are more items on my shopping list” may be true or false. If it’s true, then the action, “Purchase next item and cross it off my list” is performed. This action will be performed repeatedly while the condition remains true. The statement(s) contained in the while repetition statement constitute the body of the while. The while statement body may be a single statement or a compound statement.

Eventually, the condition will become false (when the last item on the shopping list has been purchased and crossed off the list). At this point, the repetition terminates, and the first pseudocode statement after the repetition structure is executed.

Common Programming Error 3.3
Not providing the body of a while statement with an action that eventually causes the condition in the while to become false. Normally, such a repetition structure will never terminate—an error called an “infinite loop.”

Common Programming Error 3.4
Spelling the keyword while with an uppercase W as in While (remember that C is a case-sensitive language). All of C’s reserved keywords such as while, if and else contain only lowercase letters.

As an example of an actual while, consider a program segment designed to find the first power of 3 larger than 100. Suppose the integer variable product has been initialized
to 3. When the following while repetition statement finishes executing, product will contain the desired answer:

```c
product = 3;
while ( product <= 100 ) {
    product = 3 * product;
} /* end while */
```

The flowchart of Fig. 3.4 nicely illustrates the flow of control in the while repetition statement. Once again, note that (besides small circles and arrows) the flowchart contains only a rectangle symbol and a diamond symbol. The flowchart clearly shows the repetition. The flowline emerging from the rectangle wraps back to the decision, which is tested each time through the loop until the decision eventually becomes false. At this point, the while statement is exited and control passes to the next statement in the program.

![Flowchart](image)

**Fig. 3.4** | Flowcharting the while repetition statement.

When the while statement is entered, the value of product is 3. The variable product is repeatedly multiplied by 3, taking on the values 9, 27 and 81 successively. When product becomes 243, the condition in the while statement, product <= 100, becomes false. This terminates the repetition, and the final value of product is 243. Program execution continues with the next statement after the while.

### 3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition

To illustrate how algorithms are developed, we solve several variations of a class averaging problem. Consider the following problem statement:

> A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.

The class average is equal to the sum of the grades divided by the number of students. The algorithm for solving this problem on a computer must input each of the grades, perform the averaging calculation, and print the result.

Let’s use pseudocode to list the actions to execute and specify the order in which these actions should execute. We use **counter-controlled repetition** to input the grades one at a time. This technique uses a variable called a **counter** to specify the number of times a set of statements should execute. In this example, repetition terminates when the counter exceeds 10. In this section we simply present the pseudocode algorithm (Fig. 3.5) and the corresponding C program (Fig. 3.6). In the next section we show how pseudocode algo-
3.8 Counter-Controlled Repetition

Algorithms are developed. Counter-controlled repetition is often called definite repetition because the number of repetitions is known before the loop begins executing.

---

Fig. 3.5 | Pseudocode algorithm that uses counter-controlled repetition to solve the class average problem.

```plaintext
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

---

Fig. 3.6 | C program and sample execution for the class average problem with counter-controlled repetition. (Part 1 of 2.)

```c
/* Fig. 3.6: fig03_06.c */
#include <stdio.h>

int main( void )
{
    int grade; /* grade value */
    int total; /* sum of grades input by user */
    int average; /* average of grades */

    /* initialization phase */
    total = 0; /* initialize total */
    counter = 1; /* initialize loop counter */

    /* processing phase */
    while ( counter <= 10 ) { /* loop 10 times */
        printf( "Enter grade: "); /* prompt for input */
        scanf( "%d", &grade ); /* read grade from user */
        total = total + grade; /* add grade to total */
        counter = counter + 1; /* increment counter */
    } /* end while */

    /* termination phase */
    average = total / 10; /* integer division */
    printf( "Class average is %d\n", average ); /* display result */
    return 0; /* indicate program ended successfully */
}
```

---
Note the references in the algorithm to a total and a counter. A **total** is a variable used to accumulate the sum of a series of values. A counter is a variable used to count—in this case, to count the number of grades entered. Variables used to store totals should normally be initialized to zero before being used in a program; otherwise the sum would include the previous value stored in the total’s memory location. Counter variables are normally initialized to zero or one, depending on their use (we’ll present examples showing each of these uses). An uninitialized variable contains a *garbage* value—the value last stored in the memory location reserved for that variable.

The averaging calculation in the program produced an integer result of 81. Actually, the sum of the grades in this example is 817, which when divided by 10 should yield 81.7, i.e., a number with a decimal point. We’ll see how to deal with such numbers (called floating-point numbers) in the next section.

### 3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

Let’s generalize the class average problem. Consider the following problem:

*Develop a class averaging program that will process an arbitrary number of grades each time the program is run.*

In the first class average example, the number of grades (10) was known in advance. In this example, no indication is given of how many grades are to be entered. The program must process an arbitrary number of grades. How can the program determine when to stop the input of grades? How will it know when to calculate and print the class average?
One way to solve this problem is to use a special value called a sentinel value (also called a signal value, a dummy value, or a flag value) to indicate “end of data entry.” The user types in grades until all legitimate grades have been entered. The user then types the sentinel value to indicate that the last grade has been entered. Sentinel-controlled repetition is often called indefinite repetition because the number of repetitions is not known before the loop begins executing.

Clearly, the sentinel value must be chosen so that it cannot be confused with an acceptable input value. Since grades on a quiz are normally nonnegative integers, –1 is an acceptable sentinel value for this problem. Thus, a run of the class average program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1. The program would then compute and print the class average for the grades 95, 96, 75, 74, and 89 (–1 is the sentinel value, so it should not enter into the averaging calculation).

We approach the class average program with a technique called top-down, stepwise refinement, a technique that is essential to the development of well-structured programs. We begin with a pseudocode representation of the top:

**Determine the class average for the quiz**

The top is a single statement that conveys the program’s overall function. As such, the top is, in effect, a complete representation of a program. Unfortunately, the top rarely conveys a sufficient amount of detail for writing the C program. So we now begin the refinement process. We divide the top into a series of smaller tasks and list these in the order in which they need to be performed. This results in the following first refinement.

**Initialize variables**
- Input, sum, and count the quiz grades
- Calculate and print the class average

Here, only the sequence structure has been used—the steps listed are to be executed in order, one after the other.

To proceed to the next level of refinement, i.e., the second refinement, we commit to specific variables. We need a running total of the numbers, a count of how many numbers have been processed, a variable to receive the value of each grade as it’s input and a variable to hold the calculated average. The pseudocode statement

**Initialize variables**
- Initialize total to zero
- Initialize counter to zero
Notice that only total and counter need to be initialized; the variables average and grade (for the calculated average and the user input, respectively) need not be initialized because their values will be written over by the process of destructive read-in discussed in Chapter 2. The pseudocode statement

<table>
<thead>
<tr>
<th>Input, sum, and count the quiz grades</th>
</tr>
</thead>
<tbody>
<tr>
<td>requires a repetition structure (i.e., a loop) that successively inputs each grade. Since we do not know in advance how many grades are to be processed, we’ll use sentinel-controlled repetition. The user will type legitimate grades in one at a time. After the last legitimate grade is typed, the user will type the sentinel value. The program will test for this value after each grade is input and will terminate the loop when the sentinel is entered. The refinement of the preceding pseudocode statement is then</td>
</tr>
</tbody>
</table>

| Input the first grade |
| While the user has not as yet entered the sentinel |
| Add this grade into the running total |
| Add one to the grade counter |
| Input the next grade (possibly the sentinel) |

Notice that in pseudocode, we do not use braces around the set of statements that form the body of the `while` statement. We simply indent all these statements under the `while` to show that they all belong to the `while`. Again, pseudocode is only an informal program development aid.

The pseudocode statement

<table>
<thead>
<tr>
<th>Calculate and print the class average</th>
</tr>
</thead>
<tbody>
<tr>
<td>may be refined as follows:</td>
</tr>
</tbody>
</table>

| If the counter is not equal to zero |
| Set the average to the total divided by the counter |
| Print the average |
| else |
| Print “No grades were entered” |

Notice that we’re being careful here to test for the possibility of division by zero—a fatal error that if undetected would cause the program to fail (often called “bomoming” or “crashing”). The complete second refinement is shown in Fig. 3.7.

**Common Programming Error 3.7**
An attempt to divide by zero causes a fatal error.

**Good Programming Practice 3.5**
When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program (such as printing an error message) rather than allowing the fatal error to occur.

In Fig. 3.5 and Fig. 3.7, we include some completely blank lines in the pseudocode for readability. Actually, the blank lines separate these programs into their various phases.
The pseudocode algorithm in Fig. 3.7 solves the more general class averaging problem. This algorithm was developed after only two levels of refinement. Sometimes more levels are necessary.

The C program and a sample execution are shown in Fig. 3.8. Although only integer grades are entered, the averaging calculation is likely to produce a decimal number with a decimal point. The type int cannot represent such a number. The program introduces the data type float to handle numbers with decimal points (called floating-point numbers) and introduces a special operator called a cast operator to handle the averaging calculation. These features are explained in detail after the program is presented.
/* function main begins program execution */
int main( void )
{
    int counter; /* number of grades entered */
    int grade; /* grade value */
    int total; /* sum of grades */

    float average; /* number with decimal point for average */

    /* initialization phase */
    total = 0; /* initialize total */
    counter = 0; /* initialize loop counter */

    /* processing phase */
    /* get first grade from user */
    printf( "Enter grade, -1 to end: "); /* prompt for input */
    scanf( "%d", &grade ); /* read grade from user */

    /* loop while sentinel value not yet read from user */
    while ( grade != -1 ) {
        total = total + grade; /* add grade to total */
        counter = counter + 1; /* increment counter */

        /* get next grade from user */
        printf( "Enter grade, -1 to end: "); /* prompt for input */
        scanf( "%d", &grade ); /* read next grade */
    } /* end while */

    /* termination phase */
    /* if user entered at least one grade */
    if ( counter != 0 ) {
        /* calculate average of all grades entered */
        average = ( float ) total / counter; /* avoid truncation */

        /* display average with two digits of precision */
        printf( "Class average is %.2f\n", average );
    } /* end if */
    else { /* if no grades were entered, output message */
        printf( "No grades were entered\n" );
    } /* end else */

    return 0; /* indicate program ended successfully */
} /* end function main */

Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64

Fig. 3.8 | C program and sample execution for the class average problem with sentinel-controlled repetition. (Part 2 of 3.)
Notice the compound statement in the while loop (line 24) in Fig. 3.8. Once again, the braces are necessary for all four statements to be executed within the loop. Without the braces, the last three statements in the body of the loop would fall outside the loop, causing the computer to interpret this code incorrectly as follows:

This would cause an infinite loop if the user did not input -1 for the first grade.

Averages do not always evaluate to integer values. Often, an average is a value such as 7.2 or -93.5 that contains a fractional part. These values are referred to as floating-point numbers and are represented by the data type float. The variable average is defined to be of type float (line 12) to capture the fractional result of our calculation. However, the result of the calculation total / counter is an integer because total and counter are both integer variables. Dividing two integers results in integer division in which any fractional part of the calculation is lost (i.e., truncated). Since the calculation is performed first, the fractional part is lost before the result is assigned to average. To produce a floating-point calculation with integer values, we must create temporary values that are floating-point numbers. C provides the unary cast operator to accomplish this task. Line 38

includes the cast operator (float), which creates a temporary floating-point copy of its operand, total. The value stored in total is still an integer. Using a cast operator in this manner is called explicit conversion. The calculation now consists of a floating-point value (the temporary float version of total) divided by the integer value stored in counter. Most computers can evaluate arithmetic expressions only in which the data types of the operands are identical. To ensure that the operands are of the same type, the compiler performs an operation called promotion (also called implicit conversion) on selected oper-
ands. For example, in an expression containing the data types `int` and `float`, copies of `int` operands are made and promoted to `float`. In our example, after a copy of `counter` is made and promoted to `float`, the calculation is performed and the result of the floating-point division is assigned to `average`. C provides a set of rules for promotion of operands of different types. Chapter 5 presents a discussion of all the standard data types and their order of promotion.

Cast operators are available for most data types. The cast operator is formed by placing parentheses around a data type name. The cast operator is a unary operator, i.e., an operator that takes only one operand. In Chapter 2, we studied the binary arithmetic operators. C also supports unary versions of the plus (`+`) and minus (`-`) operators, so you can write expressions like `-7` or `+5`. Cast operators associate from right to left and have the same precedence as other unary operators such as unary `+` and unary `-`. This precedence is one level higher than that of the multiplicative operators `*`, `/` and `%.

Figure 3.8 uses the `printf` conversion specifier `%.2f` (line 41) to print the value of `average`. The `f` specifies that a floating-point value will be printed. The `.2` is the precision with which the value will be displayed—with 2 digits to the right of the decimal point. If the `%f` conversion specifier is used (without specifying the precision), the default precision of 6 is used—exactly as if the conversion specifier `%.6f` had been used. When floating-point values are printed with precision, the printed value is rounded to the indicated number of decimal positions. The value in memory is unaltered. When the following statements are executed, the values 3.45 and 3.4 are printed.

```c
printf( "%.2f\n", 3.446 ); /* prints 3.45 */
printf( "%.1f\n", 3.446 ); /* prints 3.4 */
```

**Common Programming Error 3.8**
Using precision in a conversion specification in the format control string of a `scanf` statement is wrong. Precisions are used only in `printf` conversion specifications.

**Common Programming Error 3.9**
Using floating-point numbers in a manner that assumes they’re represented precisely can lead to incorrect results. Floating-point numbers are represented only approximately by most computers.

**Error-Prevention Tip 3.3**
Do not compare floating-point values for equality.

Despite the fact that floating-point numbers are not always “100% precise,” they have numerous applications. For example, when we speak of a “normal” body temperature of 98.6, we do not need to be precise to a large number of digits. When we view the temperature on a thermometer and read it as 98.6, it may actually be 98.5999473210643. The point here is that calling this number simply 98.6 is fine for most applications. We’ll say more about this issue later.

Another way floating-point numbers develop is through division. When we divide 10 by 3, the result is 3.3333333… with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so clearly the stored floating-point value can be only an approximation.
3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Structures

Let’s work another complete problem. We’ll once again formulate the algorithm using pseudocode and top-down, stepwise refinement, and write a corresponding C program. We’ve seen that control statements may be stacked on top of one another (in sequence) just as a child stacks building blocks. In this case study we’ll see the only other structured way control statements may be connected in C, namely through nesting of one control statement within another.

Consider the following problem statement:

A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, 10 of the students who completed this course took the licensing examination. Naturally, the college wants to know how well its students did on the exam. You have been asked to write a program to summarize the results. You have been given a list of these 10 students. Next to each name a 1 is written if the student passed the exam and a 2 if the student failed.

Your program should analyze the results of the exam as follows:

1. Input each test result (i.e., a 1 or a 2). Display the prompting message “Enter result” each time the program requests another test result.
2. Count the number of test results of each type.
3. Display a summary of the test results indicating the number of students who passed and the number who failed.
4. If more than eight students passed the exam, print the message “Bonus to instructor!”

After reading the problem statement carefully, we make the following observations:

1. The program must process 10 test results. A counter-controlled loop will be used.
2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine if the number is a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it’s a 2. (An exercise at the end of the chapter considers the consequences of this assumption.)
3. Two counters are used—one to count the number of students who passed the exam and one to count the number of students who failed.
4. After the program has processed all the results, it must decide if more than 8 students passed the exam.

Let’s proceed with top-down, stepwise refinement. We begin with a pseudocode representation of the top:

```
Analyze exam results and decide if instructor should receive a bonus
```

Once again, it’s important to emphasize that the top is a complete representation of the program, but several refinements are likely to be needed before the pseudocode can be naturally evolved into a C program. Our first refinement is

```
Initialize variables
Input the ten quiz grades and count passes and failures
Print a summary of the exam results and decide if instructor should receive a bonus
```
Chapter 3  Structured Program Development in C

Here, too, even though we have a complete representation of the entire program, further refinement is necessary. We now commit to specific variables. Counters are needed to record the passes and failures, a counter will be used to control the looping process, and a variable is needed to store the user input. The pseudocode statement

\[
\text{Initialize variables}
\]

may be refined as follows:

\[
\begin{align*}
\text{Initialize passes to zero} \\
\text{Initialize failures to zero} \\
\text{Initialize student to one}
\end{align*}
\]

Notice that only the counters and totals are initialized. The pseudocode statement

\[
\text{Input the ten quiz grades and count passes and failures}
\]

requires a loop that successively inputs the result of each exam. Here it’s known in advance that there are precisely ten exam results, so counter-controlled looping is appropriate. Inside the loop (i.e., nested within the loop) a double-selection statement will determine whether each exam result is a pass or a failure, and will increment the appropriate counters accordingly. The refinement of the preceding pseudocode statement is then

\[
\begin{align*}
\text{While student counter is less than or equal to ten} \\
&\quad \text{Input the next exam result} \\
&\quad \text{If the student passed} \\
&\quad\quad \text{Add one to passes} \\
&\quad\text{else} \\
&\quad\quad \text{Add one to failures} \\
&\quad \text{Add one to student counter}
\end{align*}
\]

Notice the use of blank lines to set off the \text{If…else} to improve program readability. The pseudocode statement

\[
\text{Print a summary of the exam results and decide if instructor should receive a bonus}
\]

may be refined as follows:

\[
\begin{align*}
\text{Print the number of passes} \\
\text{Print the number of failures} \\
\text{If more than eight students passed} \\
\quad \text{Print “Bonus to instructor!”}
\end{align*}
\]

The complete second refinement appears in Fig. 3.9. Notice that blank lines are also used to set off the \text{while} statement for program readability.

This pseudocode is now sufficiently refined for conversion to C. The C program and two sample executions are shown in Fig. 3.10. We’ve taken advantage of a feature of C that allows initialization to be incorporated into definitions. Such initialization occurs at compile time.
### Performance Tip 3.1

Initializing variables when they're defined can help reduce a program's execution time.

### Performance Tip 3.2

Many of the performance tips we mention in this text result in nominal improvements, so the reader may be tempted to ignore them. The cumulative effect of all these performance enhancements can make a program perform significantly faster. Also, significant improvement is realized when a supposedly nominal improvement is placed in a loop that may repeat a large number of times.

---

```c
/* Fig. 3.10: fig03_10.c */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    /* initialize variables in definitions */
    int passes = 0; /* number of passes */
    int failures = 0; /* number of failures */
    int student = 1; /* student counter */
    int result; /* one exam result */
...
```

---

**Fig. 3.9** | Pseudocode for examination results problem.

**Fig. 3.10** | C program and sample executions for examination results problem. (Part 1 of 3.)
/* process 10 students using counter-controlled loop */

while ( student <= 10 ) {

    /* prompt user for input and obtain value from user */
    printf( "Enter result ( 1=pass,2=fail ): " );
    scanf( "%d", &result );

    /* if result 1, increment passes */
    if ( result == 1 ) {
        passes = passes + 1;
    } /* end if */
    else { /* otherwise, increment failures */
        failures = failures + 1;
    } /* end else */

    student = student + 1; /* increment student counter */
} /* end while */

/* termination phase; display number of passes and failures */
printf( "Passed %d\n", passes );
printf( "Failed %d\n", failures );

/* if more than eight students passed, print "Bonus to instructor!" */
if ( passes > 8 ) {
    printf( "Bonus to instructor!\n" );
} /* end if */
return 0; /* indicate program ended successfully */
} /* end function main */

---

Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Passed 6
Failed 4

---

Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1

---

Fig. 3.10 | C program and sample executions for examination results problem. (Part 2 of 3.)
3.11 Assignment Operators

C provides several assignment operators for abbreviating assignment expressions. For example, the statement

\[ c = c + 3; \]

can be abbreviated with the **addition assignment operator** `+=` as

\[ c += 3; \]

The `+=` operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator. Any statement of the form

\[ \text{variable} = \text{variable} \ \text{operator} \ \text{expression}; \]

where `operator` is one of the binary operators `+`, `-`, `*`, `/` or `%` (or others we’ll discuss in Chapter 10), can be written in the form

\[ \text{variable} \ \text{operator} = \ \text{expression}; \]

Thus the assignment `c += 3` adds 3 to `c`. Figure 3.11 shows the arithmetic assignment operators, sample expressions using these operators and explanations.

<table>
<thead>
<tr>
<th>Assignment operator</th>
<th>Sample expression</th>
<th>Explanation</th>
<th>Assigns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Assume: int c = 3, d = 5, e = 4, f = 6, g = 12;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>+=</code></td>
<td><code>c += 7</code></td>
<td><code>c = c + 7</code></td>
<td>10 to c</td>
</tr>
<tr>
<td><code>-=</code></td>
<td><code>d -= 4</code></td>
<td><code>d = d - 4</code></td>
<td>1 to d</td>
</tr>
<tr>
<td><code>*=</code></td>
<td><code>e *= 5</code></td>
<td><code>e = e * 5</code></td>
<td>20 to e</td>
</tr>
<tr>
<td><code>/=</code></td>
<td><code>f /= 3</code></td>
<td><code>f = f / 3</code></td>
<td>2 to f</td>
</tr>
<tr>
<td><code>%=</code></td>
<td><code>g %= 9</code></td>
<td><code>g = g % 9</code></td>
<td>3 to g</td>
</tr>
</tbody>
</table>

**Fig. 3.10** | C program and sample executions for examination results problem. (Part 3 of 3.)

**Software Engineering Observation 3.6**

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working C program is normally straightforward.

**Software Engineering Observation 3.7**

Many programmers write programs without ever using program development tools such as pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs.
### 3.12 Increment and Decrement Operators

C also provides the unary increment operator, `++`, and the unary decrement operator, `--`, which are summarized in Fig. 3.12. If a variable `c` is incremented by 1, the increment operator `++` can be used rather than the expressions `c = c + 1` or `c += 1`. If increment or decrement operators are placed before a variable (i.e., prefixed), they’re referred to as the pre-increment or predecrement operators, respectively. If increment or decrement operators are placed after a variable (i.e., postfixed), they’re referred to as the postincrement or postdecrement operators, respectively. Preincrementing (predecrementing) a variable causes the variable to be incremented (decremented) by 1, then the new value of the variable is used in the expression in which it appears. Postincrementing (postdecrementing) the variable causes the current value of the variable to be used in the expression in which it appears, then the variable value is incremented (decremented) by 1.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Sample expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++</code></td>
<td><code>++a</code></td>
<td>Increment <code>a</code> by 1, then use the new value of <code>a</code> in the expression in which <code>a</code> resides.</td>
</tr>
<tr>
<td><code>++</code></td>
<td><code>a++</code></td>
<td>Use the current value of <code>a</code> in the expression in which <code>a</code> resides, then increment <code>a</code> by 1.</td>
</tr>
<tr>
<td><code>--</code></td>
<td><code>--b</code></td>
<td>Decrement <code>b</code> by 1, then use the new value of <code>b</code> in the expression in which <code>b</code> resides.</td>
</tr>
<tr>
<td><code>--</code></td>
<td><code>b--</code></td>
<td>Use the current value of <code>b</code> in the expression in which <code>b</code> resides, then decrement <code>b</code> by 1.</td>
</tr>
</tbody>
</table>

Figure 3.13 demonstrates the difference between the preincrementing and the postincrementing versions of the `++` operator. Postincrementing the variable `c` causes it to be incremented after it’s used in the `printf` statement. Preincrementing the variable `c` causes it to be incremented before it’s used in the `printf` statement.

```c
/* Fig. 3.13: fig03_13.c */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    int c; /* define variable */

    /* demonstrate postincrement */
    c = 5; /* assign 5 to c */
    printf( "%d\n", c ); /* print 5 */
    printf( "%d\n", c++ ); /* print 5 then postincrement */
    printf( "%d\n\n", c ); /* print 6 */
}
```

Figure 3.13 | Preincrementing vs. postincrementing. (Part 1 of 2.)
3.12 Increment and Decrement Operators

The program displays the value of c before and after the ++ operator is used. The decrement operator (--) works similarly.

**Good Programming Practice 3.7**

*Unary operators should be placed directly next to their operands with no intervening spaces.*

The three assignment statements in Fig. 3.10

```c
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

can be written more concisely with assignment operators as

```c
passes += 1;
failures += 1;
student += 1;
```

with preincrement operators as

```c
++passes;
++failures;
++student;
```

or with postincrement operators as

```c
passes++;
failures++;
student++;
```

It’s important to note here that when incrementing or decrementing a variable in a statement by itself, the preincrement and postincrement forms have the same effect. It’s only when a variable appears in the context of a larger expression that preincrementing and postincrementing have different effects (and similarly for predecrementing and postdecrementing). Of the expressions we’ve studied thus far, only a simple variable name may be used as the operand of an increment or decrement operator.
Chapter 3  Structured Program Development in C

Common Programming Error 3.10
Attempting to use the increment or decrement operator on an expression other than a simple variable name is a syntax error, e.g., writing ++(x + 1).

Error-Prevention Tip 3.4
C generally does not specify the order in which an operator’s operands will be evaluated (although we’ll see exceptions to this for a few operators in Chapter 4). Therefore you should avoid using statements with increment or decrement operators in which a particular variable being incremented or decremented appears more than once.

Figure 3.14 lists the precedence and associativity of the operators introduced to this point. The operators are shown top to bottom in decreasing order of precedence. The second column describes the associativity of the operators at each level of precedence. Notice that the conditional operator (? :), the unary operators increment (++), decrement (--), plus (+), minus (-) and casts, and the assignment operators =, +=, -=, *=, /= and %= associate from right to left. The third column names the various groups of operators. All other operators in Fig. 3.14 associate from left to right.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>++ (postfix)</td>
<td>right to left</td>
<td>postfix</td>
</tr>
<tr>
<td>-- (postfix)</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>+ - (type) ++ (prefix) -- (prefix)</td>
<td>right to left</td>
<td>multiplicative</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
<td>conditional</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
<td>assignment</td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
<td></td>
</tr>
<tr>
<td>+= -= *= /= %=</td>
<td>right to left</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3.14  | Precedence and associativity of the operators encountered so far in the text.

Summary

Section 3.1 Introduction
• Before writing a program to solve a particular problem, it’s essential to have a thorough understanding of the problem and a carefully planned approach to solving the problem.

Section 3.2 Algorithms
• The solution to any computing problem involves executing a series of actions in a specific order.
• A procedure for solving a problem in terms of the actions to be executed, and the order in which these actions are to be executed, is called an algorithm.
• The order in which actions are to be executed is important.

Section 3.3 Pseudocode
• Pseudocode is an artificial and informal language that helps you develop algorithms.
• Pseudocode is similar to everyday English; it’s not an actual computer programming language.
• Pseudocode programs help you “think out” a program before attempting to write it in a program-
ing language such as C.
• Pseudocode consists purely of characters; you may type pseudocode using an editor.
• Carefully prepared pseudocode programs may be converted easily to corresponding C programs.
• Pseudocode consists only of action statements.

Section 3.4 Control Structures
• Normally, statements in a program execute one after the other in the order in which they’re writ-
ten. This is called sequential execution.
• Various C statements enable you to specify that the next statement to execute may be other than
the next one in sequence. This is called transfer of control.
• Structured programming has become almost synonymous with “goto elimination.”
• Structured programs are clearer, easier to debug and modify and more likely to be bug free.
• All programs can be written in terms of only three control structures—sequence, selection and
repetition.
• Unless directed otherwise, the computer automatically executes C statements in sequence.
• A flowchart is a graphical representation of an algorithm. They’re drawn using rectangles, di-
amonds, ovals and small circles, connected by arrows called flowlines.
• The rectangle (action) symbol indicates any type of action including a calculation or an input/ output operation.
• Flowlines indicate the order in which the actions are performed.
• When drawing a flowchart that represents a complete algorithm, an oval symbol containing the
word “Begin” is the first symbol used in the flowchart; an oval symbol containing the word “End” is the last symbol used. When drawing only a portion of an algorithm, the oval symbols
are omitted in favor of using small circle symbols also called connector symbols.
• The diamond (decision) symbol indicates that a decision is to be made.
• C provides three types of selection structures in the form of statements. The if selection state-
ment either performs (selects) an action if a condition is true or skips the action if the condition
is false. The if…else selection statement performs an action if a condition is true and performs
a different action if the condition is false. The switch selection statement performs one of many
different actions depending on the value of an expression.
• The if statement is called a single-selection statement because it selects or ignores a single action.
• The if…else statement is called a double-selection statement because it selects between two dif-
terent actions.
• The switch statement is called a multiple-selection statement because it selects among many dif-
terent actions.
• C provides three types of repetition structures in the form of statements, namely while, do…while and for.
• Control statement flowchart segments can be attached to one another with control-statement stacking—connecting the exit point of one control statement to the entry point of the next.
• There is only one other way control statements may be connected—control-statement nesting.

Section 3.5 The if Selection Statement
• Selection structures are used to choose among alternative courses of action.
• The decision symbol contains an expression, such as a condition, that can be either true or false. The decision symbol has two flowlines emerging from it. One indicates the direction to be taken when the expression is true; the other indicates the direction when the expression is false.

• A decision can be based on any expression—if the expression evaluates to zero, it’s treated as false, and if the expression evaluates to nonzero, it’s treated as true.

• The if statement is a single-entry/single-exit structure.

Section 3.6 The if…else Selection Statement

• C provides the conditional operator (?:) which is closely related to the if…else statement.

• The conditional operator is C’s only ternary operator—it takes three operands. The operands together with the conditional operator form a conditional expression. The first operand is a condition. The second operand is the value for the conditional expression if the condition is true, and the third operand is the value for the conditional expression if the condition is false.

• The values in a conditional expression can also be actions to execute.

• Nested if…else statements test for multiple cases by placing if…else statements inside if…else statements.

• The if selection statement expects only one statement in its body. To include several statements in the body of an if, enclose the set of statements in braces ({}).

• A set of statements contained within a pair of braces is called a compound statement or a block.

• A syntax error is caught by the compiler. A logic error has its effect at execution time. A fatal logic error causes a program to fail and terminate prematurely. A nonfatal logic error allows a program to continue executing but to produce incorrect results.

Section 3.7 The while Repetition Statement

• The while repetition statement specifies that an action is to be repeated while a condition is true. Eventually, the condition will become false. At this point, the repetition terminates, and the first statement after the repetition statement executes.

Section 3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition

• Counter-controlled repetition uses a variable called a counter to specify the number of times a set of statements should execute.

• Counter-controlled repetition is often called definite repetition because the number of repetitions is known before the loop begins executing.

• A total is a variable used to accumulate the sum of a series of values. Variables used to store totals should normally be initialized to zero before being used in a program; otherwise the sum would include the previous value stored in the total’s memory location.

• A counter is a variable used to count. Counter variables are normally initialized to zero or one, depending on their use.

• An uninitialized variable contains a “garbage” value—the value last stored in the memory location reserved for that variable.

Section 3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

• A sentinel value (also called a signal value, a dummy value, or a flag value) is used in a sentinel-controlled loop to indicate the “end of data entry.”

• Sentinel-controlled repetition is often called indefinite repetition because the number of repetitions is not known before the loop begins executing.

• The sentinel value must be chosen so that it cannot be confused with an acceptable input value.
Summary

83

• Top-down, stepwise refinement is essential to the development of well-structured programs.
• The top is a statement that conveys the program’s overall function. It’s a complete representation of a program. The top rarely conveys a sufficient amount of detail for writing a C program. In the refinement process, we divide the top into smaller tasks and list these in execution order.
• The type float represents numbers with decimal points (called floating-point numbers).
• When dividing two integers any fractional part of the result is truncated.
• To produce a floating-point calculation with integer values, you must cast the integers to floating-point numbers. C provides the unary cast operator (float) to accomplish this task.
• Cast operators perform explicit conversions.
• Most computers can evaluate arithmetic expressions only in which the operands’ data types are identical. To ensure this, the compiler performs an operation called promotion (also called implicit conversion) on selected operands. For example, in an expression containing the data types int and float, copies of int operands are made and promoted to float.
• Cast operators are available for most data types. A cast operator is formed by placing parentheses around a data type name. The cast operator is a unary operator, i.e., it takes only one operand.
• Cast operators associate from right to left and have the same precedence as other unary operators such as unary + and unary -. This precedence is one level higher than that of *, / and %.
• The printf conversion specifier %.2f specifies that a floating-point value will be displayed with two digits to the right of the decimal point. If the %f conversion specifier is used (without specifying the precision), the default precision of 6 is used.
• When floating-point values are printed with precision, the printed value is rounded to the indicated number of decimal positions for display purposes.

Section 3.11 Assignment Operators
• C provides several assignment operators for abbreviating assignment expressions.
• The += operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator.
• Any statement of the form
  
  variable = variable operator expression;

  where operator is one of the binary operators +, -, *, / or % (or others we’ll discuss in Chapter 10), can be written in the form

  variable operator= expression;

Section 3.12 Increment and Decrement Operators
• C provides the unary increment operator, ++, and the unary decrement operator, --.
• If increment or decrement operators are placed before a variable, they’re referred to as the preincrement or predecrement operators, respectively. If increment or decrement operators are placed after a variable, they’re referred to as the postincrement or postdecrement operators, respectively.
• Preincrementing (predecrementing) a variable causes the variable to be incremented (decremented) by 1, then the new value of the variable is used in the expression in which it appears.
• Postincrementing (postdecrementing) a variable uses the current value of the variable in the expression in which it appears, then the variable value is incremented (decremented) by 1.
• When incrementing or decrementing a variable in a statement by itself, the preincrement and postincrement forms have the same effect. When a variable appears in the context of a larger expression, preincrementing and postincrementing have different effects (and similarly for predecrementing and postdecrementing).
**Terminology**

? : conditional operator 60
* : multiplication operator 72
*= : multiplication assignment operator 77
/ : division operator 72
/= : division assignment operator 77
% : remainder operator 72
%= : remainder assignment operator 77
-- : decrement operator 78
++ : increment operator 78
+= : addition assignment operator 77
-= : subtraction assignment operator 77
action : 55
action symbol : 57
addition assignment operator (+=) : 77
algorithm : 55
block : 62
"bombing" : 68
cast operator : 71
compound statement : 62
conditional expression : 60
conditional operator (?:) : 60
connector symbol : 57
control statement stacking : 58
control structure : 56
counter : 64
counter-controlled repetition : 64
"crashing" : 68
decision symbol : 57
default precision : 72
definite repetition : 65
diamond symbol : 57, 59
double-selection statement : 57
dummy value : 67
explicit conversion : 71
first refinement : 68
flag value : 67
first refinement : 68
flag value : 67
float : 69
floating-point number : 69
flowchart : 56
flowline : 57
"garbage value" : 66
goto elimination : 56
goto statement : 56
implicit conversion : 71
indefinite repetition : 67
integer division : 71
multiple-selection statement : 57
multiplicative operator : 72
nested statements : 74
nested if...else statement : 61
nesting statements : 73
order : 55
oval symbol : 57
postdecrement operator (--) : 78
postincrement operator (++) : 78
precision : 72
predecrement operator (--) : 78
preincrement operator (++) : 78
procedure : 55
program control : 55
promotion : 71
pseudocode : 55
rectangle symbol : 57
repetition statement : 63
repetition structure : 56
rounded : 72
second refinement : 67
selection structure : 56
sentinel value : 67
sequence structure : 56
sequential execution : 56
signal value : 67
single-selection statement : 57
single-entry/single-exit control statement : 58
small circle symbols : 57
top : 67
top-down, stepwise refinement : 67
total : 66
transfer of control : 56
truncated : 71
unary operator : 72
while repetition statement : 63
white-space character : 58

**Self-Review Exercises**

3.1 Fill in the blanks in each of the following questions.

a) A procedure for solving a problem in terms of the actions to be executed and the order in which the actions should be executed is called a(n) ________.

b) Specifying the execution order of statements by the computer is called ________.
c) All programs can be written in terms of three types of control statements: _______, ________, and ________.
d) The ________ selection statement is used to execute one action when a condition is true and another action when that condition is false.
e) Several statements grouped together in braces ({ and }) are called a(n) ________.
f) The ________ repetition statement specifies that a statement or group of statements is to be executed repeatedly while some condition remains true.
g) Repetition of a set of instructions a specific number of times is called ________ repetition.
h) When it’s not known in advance how many times a set of statements will be repeated, a(n) ________ value can be used to terminate the repetition.

3.2 Write four different C statements that each add 1 to integer variable x.

3.3 Write a single C statement to accomplish each of the following:
a) Assign the sum of x and y to z and increment the value of x by 1 after the calculation.
b) Multiply the variable product by 2 using the *= operator.
c) Multiply the variable product by 2 using the = and * operators.
d) Test if the value of the variable count is greater than 10. If it is, print “Count is greater than 10.”
e) Decrement the variable x by 1, then subtract it from the variable total.
f) Add the variable x to the variable total, then decrement x by 1.
g) Calculate the remainder after q is divided by divisor and assign the result to q. Write this statement two different ways.
h) Print the value 123.4567 with 2 digits of precision. What value is printed?
i) Print the floating-point value 3.14159 with three digits to the right of the decimal point. What value is printed?

3.4 Write a C statement to accomplish each of the following tasks.
a) Define variables sum and x to be of type int.
b) Initialize variable x to 1.
c) Initialize variable sum to 0.
d) Add variable x to variable sum and assign the result to variable sum.
e) Print “The sum is: “ followed by the value of variable sum.

3.5 Combine the statements that you wrote in Exercise 3.4 into a program that calculates the sum of the integers from 1 to 10. Use the while statement to loop through the calculation and increment statements. The loop should terminate when the value of x becomes 11.

3.6 Determine the values of variables product and x after the following calculation is performed. Assume that product and x each have the value 5 when the statement begins executing.
product *= x++;

3.7 Write single C statements that
a) Input integer variable x with scanf.
b) Input integer variable y with scanf.
c) Initialize integer variable i to 1.
d) Initialize integer variable power to 1.
e) Multiply variable power by x and assign the result to power.
f) Increment variable i by 1.
g) Test i to see if it’s less than or equal to y in the condition of a while statement.
h) Output integer variable power with printf.

3.8 Write a C program that uses the statements in Exercise 3.7 to calculate x raised to the y power. The program should have a while repetition control statement.
Chapter 3  Structured Program Development in C

3.9 Identify and correct the errors in each of the following:
   a)  
   ```c
   while ( c <= 5 ) {
       product *= c;
       ++c;
   }
   ```
   b)  
   ```c
   scanf( "%f", &value );
   ```
   c)  
   ```c
   if ( gender == 1 )
   printf( "Woman\n" );
   else:
   printf( "Man\n" );
   ```

3.10 What is wrong with the following while repetition statement (assume z has value 100), which is supposed to calculate the sum of the integers from 100 down to 1:
   ```c
   while ( z >= 0 )
   sum += z;
   ```

Answers to Self-Review Exercises

3.1  a) Algorithm. b) Program control. c) Sequence, selection, repetition. d) if...else. e) Compound statement. f) while. g) Counter-controlled. h) Sentinel.

3.2  
```
x = x + 1;
x += 1;
++x;
x++;
```  

3.3  
```
a)  
z = x++ + y;
b)  
product *= 2;
c)  
product = product * 2;
d)  
if ( count > 10 )
   printf( "Count is greater than 10.\n" );
e)  
total -= --x;
f)  
total += x--;  
g)  
q %= divisor;
   q = q % divisor;
h)  
printf( "%f", 123.4567 );
   123.46 is displayed.
i)  
printf( "%f\n", 3.14159 );
   3.142 is displayed.
```  

3.4  
```
a)  
int sum, x;
b)  
x = 1;
c)  
sum = 0;
d)  
sum += x; or sum = sum + x;
e)  
printf( "The sum is: %d\n", sum );
```  

3.5 See top of next page.

```
1 /* Calculate the sum of the integers from 1 to 10 */
2 #include <stdio.h>
3 4 int main( void )
5 {
6     int sum, x; /* define variables sum and x */
7```
### 3.6

Product = 25, \( x = 6 \);

### 3.7

a) \( \text{scanf}( "%d", \&x ); \)

b) \( \text{scanf}( "%d", \&y ); \)

c) \( i = 1; \)

d) \( \text{power} = 1; \)

e) \( \text{power} *= x; \)

f) \( i++; \)

g) \( \text{if} \ ( i <= y ) \)

h) \( \text{printf}( "%d", \text{power} ); \)

### 3.8

See below.

```c
/* raise x to the y power */
#include <stdio.h>

int main( void )
{
  int x, y, i, power; /* define variables */
  i = 1; /* initialize i */
  power = 1; /* initialize power */
  scanf( "%d", \&x ); /* read value for x from user */
  scanf( "%d", \&y ); /* read value for y from user */
  while ( i <= y ) { /* loop while i is less than or equal to y */
    power *= x; /* multiply power by x */
    ++i; /* increment i */
  } /* end while */
  printf( "%d", power ); /* display power */
  return 0;
} /* end main function */
```

### 3.9

a) Error: Missing the closing right brace of the while body.
   Correction: Add closing right brace after the statement ++c;

b) Error: Precision used in a scanf conversion specification.
   Correction: Remove .4 from the conversion specification.

c) Error: Semicolon after the else part of the if...else statement results in a logic error.
   The second printf will always be executed.
   Correction: Remove the semicolon after else.

### 3.10

The value of the variable z is never changed in the while statement. Therefore, an infinite loop is created. To prevent the infinite loop, z must be decremented so that it eventually becomes 0.
Exercises

3.11 Identify and correct the errors in each of the following. [Note: There may be more than one error in each piece of code.]

a) ```c
if ( age >= 65 );
    printf( "Age is greater than or equal to 65\n" );
else
    printf( "Age is less than 65\n" );
``` 

b) ```c
int x = 1, total;

while ( x <= 10 ) {
    total += x;
    ++x;
}
``` 

c) ```c
While ( x <= 100 )
    total += x;
    ++x;
``` 

d) ```c
while ( y > 0 ) {
    printf( "%d\n", y );
    ++y;
}
``` 

3.12 Fill in the blanks in each of the following:

a) The solution to any problem involves performing a series of actions in a specific ________.

b) A synonym for procedure is ________.

c) A variable that accumulates the sum of several numbers is a(n) ________.

d) Setting certain variables to specific values at the beginning of a program is called ________.

e) A special value used to indicate “end of data entry” is called a(n) ________, a(n) ________, a(n) ________ or a(n) ________ value.

f) A(n) ________ is a graphical representation of an algorithm.

g) In a flowchart, the order in which the steps should be performed is indicated by ________ symbols.

h) The termination symbol indicates the ________ and ________ of every algorithm.

i) Rectangle symbols correspond to calculations that are normally performed by statements and input/output operations that are normally performed by calls to the ________ and ________ Standard Library functions.

j) The item written inside a decision symbol is called a(n) ________.

3.13 What does the following program print?

```c
#include <stdio.h>

int main( void )
{
int x = 1, total = 0, y;

while ( x <= 10 ) {
    y = x * x;
    printf( "%d\n", y );
    total += y;
    ++x;
} /* end while */
```
3.14 Write a single pseudocode statement that indicates each of the following:
   a) Display the message "Enter two numbers".
   b) Assign the sum of variables x, y, and z to variable p.
   c) The following condition is to be tested in an if...else selection statement: The current value of variable m is greater than twice the current value of variable v.
   d) Obtain values for variables s, r, and t from the keyboard.

3.15 Formulate a pseudocode algorithm for each of the following:
   a) Obtain two numbers from the keyboard, compute their sum and display the result.
   b) Obtain two numbers from the keyboard, and determine and display which (if either) is the larger of the two numbers.
   c) Obtain a series of positive numbers from the keyboard, and determine and display their sum. Assume that the user types the sentinel value -1 to indicate “end of data entry.”

3.16 State which of the following are true and which are false. If a statement is false, explain why.
   a) Experience has shown that the most difficult part of solving a problem on a computer is producing a working C program.
   b) A sentinel value must be a value that cannot be confused with a legitimate data value.
   c) Flowlines indicate the actions to be performed.
   d) Conditions written inside decision symbols always contain arithmetic operators (i.e., +, -, *, /, and %).
   e) In top-down, stepwise refinement, each refinement is a complete representation of the algorithm.

For Exercises 3.17 to 3.21, perform each of these steps:
1. Read the problem statement.
2. Formulate the algorithm using pseudocode and top-down, stepwise refinement.
3. Write a C program.
4. Test, debug and execute the C program.

3.17 (Gas Mileage) Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several tankfuls of gasoline by recording miles driven and gallons used for each tankful. Develop a program that will input the miles driven and gallons used for each tankful. The program should calculate and display the miles per gallon obtained for each tankful. After processing all input information, the program should calculate and print the combined miles per gallon obtained for all tankfuls. Here is a sample input/output dialog:

Enter the gallons used (-1 to end): 12.8
Enter the miles driven: 287
The miles / gallon for this tank was 22.421875

Enter the gallons used (-1 to end): 10.3
Enter the miles driven: 200
The miles / gallon for this tank was 19.417475

Enter the gallons used (-1 to end): 5
Enter the miles driven: 120
The miles / gallon for this tank was 24.000000

Enter the gallons used (-1 to end): -1
The overall average miles/gallon was 21.601423
3.18 *Credit Limit Calculator* Develop a C program that will determine if a department store customer has exceeded the credit limit on a charge account. For each customer, the following facts are available:

a) Account number  
b) Balance at the beginning of the month  
c) Total of all items charged by this customer this month  
d) Total of all credits applied to this customer's account this month  
e) Allowed credit limit

The program should input each of these facts, calculate the new balance \(= \text{beginning balance} + \text{charges} - \text{credits}\), and determine if the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceeded, the program should display the customer's account number, credit limit, new balance and the message "Credit limit exceeded." Here is a sample input/output dialog:

```
Enter account number (-1 to end): 100  
Enter beginning balance: 5394.78  
Enter total charges: 1000.00  
Enter total credits: 500.00  
Enter credit limit: 5500.00  
Account: 100  
Balance: 5894.78  
Credit Limit Exceeded.  
```

```
Enter account number (-1 to end): 200  
Enter beginning balance: 1000.00  
Enter total charges: 123.45  
Enter total credits: 321.00  
Enter credit limit: 1500.00  
```

```
Enter account number (-1 to end): 300  
Enter beginning balance: 500.00  
Enter total charges: 274.73  
Enter total credits: 100.00  
Enter credit limit: 800.00  
```

```
Enter account number (-1 to end): -1  
```

3.19 *Sales Commission Calculator* One large chemical company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9% of their gross sales for that week. For example, a salesperson who sells $5000 worth of chemicals in a week receives $200 plus 9% of $5000, or a total of $650. Develop a program that will input each salesperson's gross sales for last week and will calculate and display that salesperson's earnings. Process one salesperson's figures at a time. Here is a sample input/output dialog:

```
Enter sales in dollars (-1 to end): 5000.00  
Salary is: $650.00  
Enter sales in dollars (-1 to end): 1234.56  
Salary is: $311.11  
Enter sales in dollars (-1 to end): 1088.89  
Salary is: $298.00  
Enter sales in dollars (-1 to end): -1  
```

3.20 *Interest Calculator* The simple interest on a loan is calculated by the formula

\[ \text{interest} = \text{principal} \times \text{rate} \times \text{days} / 365; \]
The preceding formula assumes that rate is the annual interest rate, and therefore includes the division by 365 (days). Develop a program that will input principal, rate and days for several loans, and will calculate and display the simple interest for each loan, using the preceding formula. Here is a sample input/output dialog:

Enter loan principal (-1 to end): 1000.00
Enter interest rate: .1
Enter term of the loan in days: 365
The interest charge is $100.00

Enter loan principal (-1 to end): 1000.00
Enter interest rate: .08375
Enter term of the loan in days: 224
The interest charge is $51.40

Enter loan principal (-1 to end): 10000.00
Enter interest rate: .09
Enter term of the loan in days: 1460
The interest charge is $3600.00

Enter loan principal (-1 to end): -1

3.21  (Salary Calculator) Develop a program that will determine the gross pay for each of several employees. The company pays “straight time” for the first 40 hours worked by each employee and pays “time-and-a-half” for all hours worked in excess of 40 hours. You’re given a list of the employees of the company, the number of hours each employee worked last week and the hourly rate of each employee. Your program should input this information for each employee, and should determine and display the employee’s gross pay. Here is a sample input/output dialog:

Enter # of hours worked (-1 to end): 39
Enter hourly rate of the worker ($00.00): 10.00
Salary is $390.00

Enter # of hours worked (-1 to end): 40
Enter hourly rate of the worker ($00.00): 10.00
Salary is $400.00

Enter # of hours worked (-1 to end): 41
Enter hourly rate of the worker ($00.00): 10.00
Salary is $415.00

Enter # of hours worked (-1 to end): -1

3.22  (Predecrementing vs. Postdecrementing) Write a program that demonstrates the difference between predecrementing and postdecrementing using the decrement operator --.

3.23  (Printing Numbers from a Loop) Write a program that utilizes looping to print the numbers from 1 to 10 side by side on the same line with three spaces between numbers.

3.24  (Find the Largest Number) The process of finding the largest number (i.e., the maximum of a group of numbers) is used frequently in computer applications. For example, a program that determines the winner of a sales contest would input the number of units sold by each salesperson. The salesperson who sold the most units wins the contest. Write a pseudocode program and then a program that inputs a series of 10 numbers and determines and prints the largest of the numbers. [Hint: Your program should use three variables as follows]:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter</td>
<td>A counter to count to 10 (i.e., to keep track of how many numbers have been input and to determine when all 10 numbers have been processed)</td>
</tr>
<tr>
<td>number</td>
<td>The current number input to the program</td>
</tr>
<tr>
<td>largest</td>
<td>The largest number found so far</td>
</tr>
</tbody>
</table>
Section 3.25 (Tabular Output) Write a program that uses looping to print the following table of values. Use the tab escape sequence, \t, in the printf statement to separate the columns with tabs.

<table>
<thead>
<tr>
<th>N</th>
<th>10*N</th>
<th>100*N</th>
<th>1000*N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>200</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>300</td>
<td>3000</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>400</td>
<td>4000</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>500</td>
<td>5000</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td>600</td>
<td>6000</td>
</tr>
<tr>
<td>7</td>
<td>70</td>
<td>700</td>
<td>7000</td>
</tr>
<tr>
<td>8</td>
<td>80</td>
<td>800</td>
<td>8000</td>
</tr>
<tr>
<td>9</td>
<td>90</td>
<td>900</td>
<td>9000</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>10000</td>
</tr>
</tbody>
</table>

Section 3.26 (Tabular Output) Write a program that utilizes looping to produce the following table of values:

<table>
<thead>
<tr>
<th>A</th>
<th>A+2</th>
<th>A+4</th>
<th>A+6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>15</td>
<td>17</td>
<td>19</td>
<td>21</td>
</tr>
</tbody>
</table>

Section 3.27 (Find the Two Largest Numbers) Using an approach similar to Exercise 3.24, find the two largest values of the 10 numbers. [Note: You may input each number only once.]

Section 3.28 (Validating User Input) Modify the program in Figure 3.10 to validate its inputs. On any input, if the value entered is other than 1 or 2, keep looping until the user enters a correct value.

Section 3.29 What does the following program print?

```c
#include <stdio.h>
int main( void )
{
    int count = 1; /* initialize count */

    while ( count <= 10 ) { /* loop 10 times */
        /* output line of text */
        printf( "%s
", count % 2 ? "****" : "++++++++" );
        count++; /* increment count */
    } /* end while */

    return 0; /* indicate program ended successfully */
} /* end function main */
```

Section 3.30 What does the following program print?

```c
#include <stdio.h>
int main( void )
{
    int row = 10; /* initialize row */
    int column; /* define column */
```
3.31  *(Dangling Else Problem)* Determine the output for each of the following when \(x\) is 9 and \(y\) is 11, and when \(x\) is 11 and \(y\) is 9. The compiler ignores the indentation in a C program. Also, the compiler always associates an `else` with the previous `if` unless told to do otherwise by the placement of braces `{}`. Because, on first glance, you may not be sure which `if` an `else` matches, this is referred to as the “dangling else” problem. We eliminated the indentation from the following code to make the problem more challenging. [Hint: Apply indentation conventions you have learned.]

a)  
```c
if ( x < 10 )
    if ( y > 10 )
        printf( "*****\n" );
    else
        printf( "#####\n" );
        printf( "$$$$$\n" );
```

b)  
```c
if ( x < 10 ) {
    if ( y > 10 )
        printf( "*****\n" );
} else {
    printf( "#####\n" );
    printf( "$$$$$\n" );
}
```

3.32  *(Another Dangling Else Problem)* Modify the following code to produce the output shown. Use proper indentation techniques. You may not make any changes other than inserting braces. The compiler ignores the indentation in a program. We eliminated the indentation from the following code to make the problem more challenging. [Note: It’s possible that no modification is necessary.]

```c
if ( y == 8 )
    if ( x == 5 )
        printf( "@@@@@
" );
    else
        printf( "#####\n" );
        printf( "$$$$$\n" );
        printf( "&&&&&\n" );
```

a)  Assuming \(x = 5\) and \(y = 8\), the following output is produced.
Chapter 3  Structured Program Development in C

b) Assuming \( x = 5 \) and \( y = 8 \), the following output is produced.

```
********
```

c) Assuming \( x = 5 \) and \( y = 8 \), the following output is produced.

```
########
```

d) Assuming \( x = 5 \) and \( y = 7 \), the following output is produced. \([\text{Note: The last three printf statements are all part of a compound statement.}]\)

```
####
$\ldots\ldots$
```

3.33  \((\text{Square of Asterisks})\) Write a program that reads in the side of a square and then prints that square out of asterisks. Your program should work for squares of all side sizes between 1 and 20. For example, if your program reads a size of 4, it should print

```
****
****
****
****
```

3.34  \((\text{Hollow Square of Asterisks})\) Modify the program you wrote in Exercise 3.33 so that it prints a hollow square. For example, if your program reads a size of 5, it should print

```
*****
* *
* *
* *
*****
```

3.35  \((\text{Palindrome Tester})\) A palindrome is a number or a text phrase that reads the same backward as forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a program that reads in a five-digit integer and determines whether or not it’s a palindrome. \([\text{Hint: Use the division and remainder operators to separate the number into its individual digits.}]\)

3.36  \((\text{Printing the Decimal Equivalent of a Binary Number})\) Input an integer containing only 0s and 1s (i.e., a “binary” integer) and print its decimal equivalent. \([\text{Hint: Use the remainder and division operators to pick off the “binary” number’s digits one at a time from right to left. Just as in the decimal number system, in which the rightmost digit has a positional value of 1, and the next digit left has a positional value of 10, then 100, then 1000, and so on, in the binary number system the rightmost digit has a positional value of 1, the next digit left has a positional value of 2, then 4, then 8, and so on. Thus the decimal number 234 can be interpreted as 4 \times 1 + 3 \times 10 + 2 \times 100. The decimal equivalent of binary 1101 is 1 \times 1 + 0 \times 2 + 1 \times 4 + 1 \times 8 or 101 + 0 + 4 + 8 or 13.}]\)

3.37  \((\text{How Fast is Your Computer?})\) How can you determine how fast your own computer really operates? Write a program with a \texttt{while} loop that counts from 1 to 300,000,000 by 1s. Every time the count reaches a multiple of 100,000,000, print that number on the screen. Use your watch to time how long each 100 million repetitions of the loop takes.
3.38 Write a program that prints 100 asterisks, one at a time. After every tenth asterisk, your program should print a newline character. [Hint: Count from 1 to 100. Use the remainder operator to recognize each time the counter reaches a multiple of 10.]

3.39 (Counting 7s) Write a program that reads an integer and determines and prints how many digits in the integer are 7s.

3.40 (Checkerboard Pattern of Asterisks) Write a program that displays the following checkerboard pattern:

```
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
```

Your program must use only three output statements, one of each of the following forms:

```c
printf( "\n" );
printf( "\n" );
printf( "\n" );
```

3.41 (Multiples of 2 with an Infinite Loop) Write a program that keeps printing the multiples of the integer 2, namely 2, 4, 8, 16, 32, 64, and so on. Your loop should not terminate (i.e., you should create an infinite loop). What happens when you run this program?

3.42 (Diameter, Circumference and Area of a Circle) Write a program that reads the radius of a circle (as a `float` value) and computes and prints the diameter, the circumference and the area. Use the value 3.14159 for π.

3.43 What is wrong with the following statement? Rewrite the statement to accomplish what the programmer was probably trying to do.

```c
printf( "%d", ++( x + y ) );
```

3.44 (Sides of a Triangle) Write a program that reads three nonzero `float` values and determines and prints if they could represent the sides of a triangle.

3.45 (Sides of a Right Triangle) Write a program that reads three nonzero integers and determines and prints if they could be the sides of a right triangle.

3.46 (Factorial) The factorial of a nonnegative integer n is written n! (pronounced “n factorial”) and is defined as follows:

\[ n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1 \]  
(for values of n greater than or equal to 1)

and

\[ 0! = 1 \]  
(for n = 0).

For example, 5! = 5 · 4 · 3 · 2 · 1, which is 120.

a) Write a program that reads a nonnegative integer and computes and prints its factorial.

b) Write a program that estimates the value of the mathematical constant e by using the formula:

\[ e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots \]

c) Write a program that computes the value of \(e^x\) by using the formula

\[ e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \]
Making a Difference

3.47 (World Population Growth Calculator) Use the web to determine the current world population and the annual world population growth rate. Write an application that inputs these values, then displays the estimated world population after one, two, three, four and five years.

3.48 (Target-Heart-Rate Calculator) While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your trainers and doctors. According to the American Heart Association (AHA) (www.americanheart.org/presenter.jhtml?identifier=4736), the formula for calculating your maximum heart rate in beats per minute is 220 minus your age in years. Your target heart rate is a range that is 50–85% of your maximum heart rate. [Note: These formulas are estimates provided by the AHA. Maximum and target heart rates may vary based on the health, fitness and gender of the individual. Always consult a physician or qualified health care professional before beginning or modifying an exercise program.] Create a program that reads the user’s birthday and the current day (each consisting of the month, day and year). Your program should calculate and display the person’s age (in years), the person’s maximum heart rate and the person’s target heart rate range.

3.49 (Enforcing Privacy with Cryptography) The explosive growth of Internet communications and data storage on Internet-connected computers has greatly increased privacy concerns. The field of cryptography is concerned with coding data to make it difficult (and hopefully—with the most advanced schemes—impossible) for unauthorized users to read. In this exercise you’ll investigate a simple scheme for encrypting and decrypting data. A company that wants to send data over the Internet has asked you to write a program that will encrypt it so that it may be transmitted more securely. All the data is transmitted as four-digit integers. Your application should read a four-digit integer entered by the user and encrypt it as follows: Replace each digit with the result of adding 7 to the digit and getting the remainder after dividing the new value by 10. Then swap the first digit with the third, and swap the second digit with the fourth. Then print the encrypted integer. Write a separate application that inputs an encrypted four-digit integer and decrypts it (by reversing the encryption scheme) to form the original number. [Optional reading project: Research “public key cryptography” in general and the PGP (Pretty Good Privacy) specific public key scheme. You may also want to investigate the RSA scheme, which is widely used in industrial-strength applications.]
4

C Program Control

Not everything that can be counted counts, and not every thing that counts can be counted.
—Albert Einstein

Who can control his fate?
—William Shakespeare

The used key is always bright.
—Benjamin Franklin

Every advantage in the past is judged in the light of the final issue.
—Demosthenes

Objectives
In this chapter, you’ll learn:

- The essentials of counter-controlled repetition.
- To use the for and do...while repetition statements to execute statements repeatedly.
- To understand multiple selection using the switch selection statement.
- To use the break and continue statements to alter the flow of control.
- To use the logical operators to form complex conditional expressions in control statements.
- To avoid the consequences of confusing the equality and assignment operators.
4.1 Introduction
You should now be comfortable with writing simple but complete C programs. In this chapter, repetition is considered in greater detail, and additional repetition control statements, namely the for and the do...while, are presented. The switch multiple-selection statement is introduced. We discuss the break statement for exiting immediately from certain control statements, and the continue statement for skipping the remainder of the body of a repetition statement and proceeding with the next iteration of the loop. The chapter discusses logical operators used for combining conditions, and summarizes the principles of structured programming as presented in Chapter 3 and 4.

4.2 Repetition Essentials
Most programs involve repetition, or looping. A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true. We have discussed two means of repetition:

1. Counter-controlled repetition
2. Sentinel-controlled repetition

Counter-controlled repetition is sometimes called definite repetition because we know in advance exactly how many times the loop will be executed. Sentinel-controlled repetition is sometimes called indefinite repetition because it’s not known in advance how many times the loop will be executed.

In counter-controlled repetition, a control variable is used to count the number of repetitions. The control variable is incremented (usually by 1) each time the group of instructions is performed. When the value of the control variable indicates that the correct number of repetitions has been performed, the loop terminates and the computer continues executing with the statement after the repetition statement.

Sentinel values are used to control repetition when:

1. The precise number of repetitions is not known in advance, and
2. The loop includes statements that obtain data each time the loop is performed.

The sentinel value indicates “end of data.” The sentinel is entered after all regular data items have been supplied to the program. Sentinels must be distinct from regular data items.
4.3 Counter-Controlled Repetition

Counter-controlled repetition requires:

1. The name of a control variable (or loop counter).
2. The initial value of the control variable.
3. The increment (or decrement) by which the control variable is modified each time through the loop.
4. The condition that tests for the final value of the control variable (i.e., whether looping should continue).

Consider the simple program shown in Fig. 4.1, which prints the numbers from 1 to 10. The definition

```
int counter = 1; /* initialization */
```

names the control variable (counter), defines it to be an integer, reserves memory space for it, and sets it to an initial value of 1. This definition is not an executable statement.

```c
/* Fig. 4.1: fig04_01.c */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    int counter = 1; /* initialization */
    while ( counter <= 10 ) { /* repetition condition */
        printf ( "%d\n", counter ); /* display counter */
        ++counter; /* increment */
    } /* end while */
    return 0; /* indicate program ended successfully */
} /* end function main */
```

Fig. 4.1  |  Counter-controlled repetition.

The definition and initialization of counter could also have been written as

```
int counter;
counter = 1;
```
The definition is not executable, but the assignment is. We use both methods of initializing variables.

The statement

```c
++counter; /* increment */
```

increments the loop counter by 1 each time the loop is performed. The loop-continuation condition in the `while` statement tests if the value of the control variable is less than or equal to 10 (the last value for which the condition is true). The body of this `while` is performed even when the control variable is 10. The loop terminates when the control variable exceeds 10 (i.e., `counter` becomes 11).

You could make the program in Fig. 4.1 more concise by initializing `counter` to 0 and by replacing the `while` statement with

```c
while ( ++counter <= 10 )
    printf( "%d\n", counter );
```

This code saves a statement because the incrementing is done directly in the `while` condition before the condition is tested. Also, this code eliminates the need for the braces around the body of the `while` because the `while` now contains only one statement. Coding in such a condensed fashion takes some practice. Some programmers feel that this makes the code too cryptic and error prone.

**Common Programming Error 4.1**

Floating-point values may be approximate, so controlling counting loops with floating-point variables may result in imprecise counter values and inaccurate termination tests.

**Error-Prevention Tip 4.1**

Control counting loops with integer values.

**Good Programming Practice 4.1**

Too many levels of nesting can make a program difficult to understand. As a rule, try to avoid using more than three levels of nesting.

**Good Programming Practice 4.2**

The combination of vertical spacing before and after control statements and indentation of the bodies of control statements within the control-statement headers gives programs a two-dimensional appearance that greatly improves program readability.

### 4.4 for Repetition Statement

The `for` repetition statement handles all the details of counter-controlled repetition. To illustrate its power, let’s rewrite the program of Fig. 4.1. The result is shown in Fig. 4.2.

```c
/* Fig. 4.2: fig04_02.c */
Counter-controlled repetition with the `for` statement */
#include <stdio.h>
```

Fig. 4.2 | Counter-controlled repetition with the `for` statement. (Part 1 of 2.)
The program operates as follows. When the `for` statement begins executing, the control variable `counter` is initialized to 1. Then, the loop-continuation condition `counter <= 10` is checked. Because the initial value of `counter` is 1, the condition is satisfied, so the `printf` statement (line 13) prints the value of `counter`, namely 1. The control variable `counter` is then incremented by the expression `counter++`, and the loop begins again with the loop-continuation test. Since the control variable is now equal to 2, the final value is not exceeded, so the program performs the `printf` statement again. This process continues until the control variable `counter` is incremented to its final value of 11—this causes the loop-continuation test to fail, and repetition terminates. The program continues by performing the first statement after the `for` statement (in this case, the `return` statement at the end of the program).

Figure 4.3 takes a closer look at the `for` statement of Fig. 4.2. Notice that the `for` statement “does it all”—it specifies each of the items needed for counter-controlled repetition with a control variable. If there is more than one statement in the body of the `for`, braces are required to define the body of the loop.

Notice that Fig. 4.2 uses the loop-continuation condition `counter <= 10`. If you incorrectly wrote `counter < 10`, then the loop would be executed only 9 times. This is a common logic error called an off-by-one error.
The general format of the `for` statement is

```
for ( expression1 ; expression2 ; expression3 )
  statement
```

where `expression1` initializes the loop-control variable, `expression2` is the loop-continuation condition, and `expression3` increments the control variable. In most cases, the `for` statement can be represented with an equivalent `while` statement as follows:

```
expression1;
while ( expression2 ) {
  statement
  expression3;
}
```

There is an exception to this rule, which we discuss in Section 4.9.

Often, `expression1` and `expression3` are comma-separated lists of expressions. The commas as used here are actually **comma operators** that guarantee that lists of expressions evaluate from left to right. The value and type of a comma-separated list of expressions are the value and type of the right-most expression in the list. The comma operator is most often used in the `for` statement. Its primary use is to enable you to use multiple initialization and/or multiple increment expressions. For example, there may be two control variables in a single `for` statement that must be initialized and incremented.

```
counter = counter + 1
counter += 1
++counter
counter++
```

The three expressions in the `for` statement are optional. If `expression2` is omitted, C assumes that the condition is true, thus creating an infinite loop. One may omit `expression1` if the control variable is initialized elsewhere in the program. `expression3` may be omitted if the increment is calculated by statements in the body of the `for` statement or if no increment is needed. The increment expression in the `for` statement acts like a stand-alone C statement at the end of the body of the `for`. Therefore, the expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```
are all equivalent in the increment part of the for statement. Many C programmers prefer the form `counter++` because the incrementing occurs after the loop body is executed, and the postincrementing form seems more natural. Because the variable being preincremented or postincremented here does not appear in a larger expression, both forms of incrementing have the same effect. The two semicolons in the for statement are required.

4.5 **for Statement: Notes and Observations**

1. The initialization, loop-continuation condition and increment can contain arithmetic expressions. For example, if \( x = 2 \) and \( y = 10 \), the statement

   ```c
   for ( j = x; j <= 4 * x * y; j += y / x )
   ```

   is equivalent to the statement

   ```c
   for ( j = 2; j <= 80; j += 5 )
   ```

2. The “increment” may be negative (in which case it’s really a decrement and the loop actually counts downward).

3. If the loop-continuation condition is initially false, the loop body does not execute. Instead, execution proceeds with the statement following the for statement.

4. The control variable is frequently printed or used in calculations in the body of a loop, but it need not be. It’s common to use the control variable for controlling repetition while never mentioning it in the body of the loop.

5. The for statement is flowcharted much like the while statement. For example, Fig. 4.4 shows the flowchart of the for statement

   ```c
   for ( counter = 1; counter <= 10; counter++ )
   printf( "%d", counter );
   ```

   This flowchart makes it clear that the initialization occurs only once and that incrementing occurs after the body statement is performed.

**Common Programming Error 4.3**

*Using commas instead of semicolons in a for header is a syntax error.*

**Common Programming Error 4.4**

*Placing a semicolon immediately to the right of a for header makes the body of that for statement an empty statement. This is normally a logic error.*

**Error-Prevention Tip 4.3**

*Although the value of the control variable can be changed in the body of a for loop, this can lead to subtle errors. It's best not to change it.*

4.6 **Examples Using the for Statement**

The following examples show methods of varying the control variable in a for statement.

1. Vary the control variable from 1 to 100 in increments of 1.

   ```c
   for ( i = 1; i <= 100; i++ )
   ```
2. Vary the control variable from 100 to 1 in increments of -1 (decrements of 1).

   \[
   \textbf{for } \ ( i = 100; \ i \geq 1; \ i-- )
   \]

3. Vary the control variable from 7 to 77 in steps of 7.

   \[
   \textbf{for } \ ( i = 7; \ i \leq 77; \ i += 7 )
   \]

4. Vary the control variable from 20 to 2 in steps of -2.

   \[
   \textbf{for } \ ( i = 20; \ i \geq 2; \ i -= 2 )
   \]

5. Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.

   \[
   \textbf{for } \ ( j = 2; \ j \leq 17; \ j += 3 )
   \]

6. Vary the control variable over the following sequence of values: 44, 33, 22, 11, 0.

   \[
   \textbf{for } \ ( j = 44; \ j \geq 0; \ j -= 11 )
   \]

The next two examples provide simple applications of the\textbf{ for} statement. Figure 4.5
uses the\textbf{ for} statement to sum all the even integers from 2 to 100.

```
/* Fig. 4.5: fig04_05.c
Summation with for */
#include <stdio.h>
int main( void )
{
    int sum = 0; /* initialize sum */
    int number; /* number to be added to sum */
    for ( number = 2; number <= 100; number += 2 ) {
        sum += number; /* add number to sum */
    } /* end for */
```

Fig. 4.4 | Flowcharting a typical for repetition statement.
The body of the `for` statement in Fig. 4.5 could actually be merged into the rightmost portion of the `for` header by using the comma operator as follows:

```
for ( number = 2; number <= 100; sum += number, number += 2 )
/* empty statement */
```

The initialization `sum = 0` could also be merged into the initialization section of the `for`.

**Good Programming Practice 4.3**

Although statements preceding a `for` and statements in the body of a `for` can often be merged into the `for` header, avoid doing so because it makes the program more difficult to read.

**Good Programming Practice 4.4**

Limit the size of control-statement headers to a single line if possible.

The next example computes compound interest using the `for` statement. Consider the following problem statement:

*A person invests $1000.00 in a savings account yielding 5% interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:

\[ a = p(1 + r)^n \]

where

- \( p \) is the original amount invested (i.e., the principal)
- \( r \) is the annual interest rate
- \( n \) is the number of years
- \( a \) is the amount on deposit at the end of the \( n \)th year.*

This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit. The solution is shown in Fig. 4.6.
Chapter 4  C Program Control

The for statement executes the body of the loop 10 times, varying a control variable from 1 to 10 in increments of 1. Although C does not include an exponentiation operator, we can use the Standard Library function pow for this purpose. The function pow(x, y) calculates the value of x raised to the yth power. It takes two arguments of type double and returns a double value. Type double is a floating-point type much like float, but typically a variable of type double can store a value of much greater magnitude with greater precision than float. The header <math.h> (line 4) should be included whenever a math function such as pow is used. Actually, this program would malfunction without the inclusion of math.h, as the linker would be unable to find the pow function. Function pow requires

---

1. On many Linux/UNIX C compilers, you must include the -lm option (e.g., cc -lm fig04_06.c) when compiling Fig. 4.6. This links the math library to the program.
two double arguments, but variable year is an integer. The math.h file includes information that tells the compiler to convert the value of year to a temporary double representation before calling the function. This information is contained in something called pow’s function prototype. Function prototypes are explained in Chapter 5. We also provide a summary of the pow function and other math library functions in Chapter 5.

Notice that we defined the variables amount, principal and rate to be of type double. We did this for simplicity because we’re dealing with fractional parts of dollars.

Error-Prevention Tip 4.4
Do not use variables of type float or double to perform monetary calculations. The imprecision of floating-point numbers can cause errors that will result in incorrect monetary values. [In this chapter’s exercises, we explore the use of integers to perform monetary calculations.]

Here is a simple explanation of what can go wrong when using float or double to represent dollar amounts. Two float dollar amounts stored in the machine could be 14.234 (which with %.2f prints as 14.23) and 18.673 (which with %.2f prints as 18.67). When these amounts are added, they produce the sum 32.907, which with %.2f prints as 32.91. Thus your printout could appear as

```
14.23  
+ 18.67 
------- 
32.91
```

Clearly the sum of the individual numbers as printed should be 32.90! You’ve been warned!

The conversion specifier %21.2f is used to print the value of the variable amount in the program. The 21 in the conversion specifier denotes the field width in which the value will be printed. A field width of 21 specifies that the value printed will appear in 21 print positions. The 2 specifies the precision (i.e., the number of decimal positions). If the number of characters displayed is less than the field width, then the value will automatically be right justified in the field. This is particularly useful for aligning floating-point values with the same precision (so that their decimal points align vertically). To left justify a value in a field, place a - (minus sign) between the % and the field width. The minus sign may also be used to left justify integers (such as in %-6d) and character strings (such as in %-8s). We’ll discuss the powerful formatting capabilities of printf and scanf in detail in Chapter 9.

4.7 switch Multiple-Selection Statement

In Chapter 3, we discussed the if single-selection statement and the if…else double-selection statement. Occasionally, an algorithm will contain a series of decisions in which a variable or expression is tested separately for each of the constant integral values it may assume, and different actions are taken. This is called multiple selection. C provides the switch multiple-selection statement to handle such decision making.

The switch statement consists of a series of case labels, an optional default case and statements to execute for each case. Figure 4.7 uses switch to count the number of each different letter grade students earned on an exam.
/* Fig. 4.7: fig04_07.c 
Counting letter grades */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    int grade; /* one grade */
    int aCount = 0; /* number of As */
    int bCount = 0; /* number of Bs */
    int cCount = 0; /* number of Cs */
    int dCount = 0; /* number of Ds */
    int fCount = 0; /* number of Fs */

    printf( "Enter the letter grades.\n" );
    printf( "Enter the EOF character to end input.\n" );

    /* loop until user types end-of-file key sequence */
    while ( ( grade = getchar() ) != EOF )
    {
        /* determine which grade was input */
        switch ( grade ) { /* switch nested in while */

            case 'A': /* grade was uppercase A */
            case 'a': /* or lowercase a */
                ++aCount; /* increment aCount */
                break; /* necessary to exit switch */

            case 'B': /* grade was uppercase B */
            case 'b': /* or lowercase b */
                ++bCount; /* increment bCount */
                break; /* exit switch */

            case 'C': /* grade was uppercase C */
            case 'c': /* or lowercase c */
                ++cCount; /* increment cCount */
                break; /* exit switch */

            case 'D': /* grade was uppercase D */
            case 'd': /* or lowercase d */
                ++dCount; /* increment dCount */
                break; /* exit switch */

            case 'F': /* grade was uppercase F */
            case 'f': /* or lowercase f */
                ++fCount; /* increment fCount */
                break; /* exit switch */

            case '\n': /* ignore newlines, */
            case '\t': /* tabs, */
            case ' ': /* and spaces in input */
                break; /* exit switch */

Fig. 4.7 | switch example. (Part 1 of 2.)
In the program, the user enters letter grades for a class. In the while header (line 19), the parenthesized assignment (grade = getchar()) executes first. The getchar function (from <stdio.h>) reads one character from the keyboard and stores that character in the integer variable grade. Characters are normally stored in variables of type char. However, an important feature of C is that characters can be stored in any integer data type because they're usually represented as one-byte integers in the computer. Thus, we can treat a character as either an integer or a character, depending on its use. For example, the statement

```c
printf( "The character (%c) has the value %d.\n", 'a', 'a' );
```

Fig. 4.7 | switch example. (Part 2 of 2.)
uses the conversion specifiers %c and %d to print the character a and its integer value, respectively. The result is

| The character (a) has the value 97. |

The integer 97 is the character’s numerical representation in the computer. Many computers today use the ASCII (American Standard Code for Information Interchange) character set in which 97 represents the lowercase letter ‘a’. A list of the ASCII characters and their decimal values is presented in Appendix B. Characters can be read with scanf by using the conversion specifier %c.

Assignments as a whole actually have a value. This value is assigned to the variable on the left side of the =. The value of the assignment expression grade = getchar() is the character that is returned by getchar and assigned to the variable grade.

The fact that assignments have values can be useful for setting several variables to the same value. For example,

```
a = b = c = 0;
```

first evaluates the assignment c = 0 (because the = operator associates from right to left). The variable b is then assigned the value of the assignment c = 0 (which is 0). Then, the variable a is assigned the value of the assignment b = (c = 0) (which is also 0). In the program, the value of the assignment grade = getchar() is compared with the value of EOF (a symbol whose acronym stands for “end of file”). We use EOF (which normally has the value -1) as the sentinel value. The user types a system-dependent keystroke combination to mean “end of file”—i.e., “I have no more data to enter.” EOF is a symbolic integer constant defined in the <stdio.h> header (we’ll see how symbolic constants are defined in Chapter 6). If the value assigned to grade is equal to EOF, the program terminates. We have chosen to represent characters in this program as ints because EOF has an integer value (again, normally -1).

---

**Portability Tip 4.1**

The keystroke combinations for entering EOF (end of file) are system dependent.

**Portability Tip 4.2**

Testing for the symbolic constant EOF rather than -1 makes programs more portable. The C standard states that EOF is a negative integral value (but not necessarily -1). Thus, EOF could have different values on different systems.

On Linux/UNIX/Mac OS X systems, the EOF indicator is entered by typing

```
<Ctrl> d
```

on a line by itself. This notation <Ctrl> d means to press the Enter key and then simultaneously press both the Ctrl key and the d key. On other systems, such as Microsoft Windows, the EOF indicator can be entered by typing

```
<Ctrl> z
```

You may also need to press Enter on Windows.

The user enters grades at the keyboard. When the Enter key is pressed, the characters are read by function getchar one character at a time. If the character entered is not equal
to EOF, the switch statement (line 22) is entered. Keyword switch is followed by the variable name grade in parentheses. This is called the controlling expression. The value of this expression is compared with each of the case labels. Assume the user has entered the letter C as a grade. C is automatically compared to each case in the switch. If a match occurs (case 'C'), the statements for that case are executed. In the case of the letter C, cCount is incremented by 1 (line 36), and the switch statement is exited immediately with the break statement.

The break statement causes program control to continue with the first statement after the switch statement. The break statement is used because the cases in a switch statement would otherwise run together. If break is not used anywhere in a switch statement, then each time a match occurs in the statement, the statements for all the remaining cases will be executed. (This feature is rarely useful, although it’s perfect for programming the iterative song *The Twelve Days of Christmas!* If no match occurs, the default case is executed, and an error message is printed.

Each case can have one or more actions. The switch statement is different from all other control statements in that braces are not required around multiple actions in a case of a switch. The general switch multiple-selection statement (using a break in each case) is flowcharted in Fig. 4.8. The flowchart makes it clear that each break statement at the end of a case causes control to immediately exit the switch statement.

---

Fig. 4.8 | switch multiple-selection statement with breaks.
In the switch statement of Fig. 4.7, the lines

```c
  case '\n': /* ignore newlines, */
  case '\t': /* tabs, */
  case ' ': /* and spaces in input */
  break; /* exit switch */
```

cause the program to skip newline, tab and blank characters. Reading characters one at a time can cause some problems. To have the program read the characters, they must be sent to the computer by pressing the Enter key. This causes the newline character to be placed in the input after the character we wish to process. Often, this newline character must be specially processed to make the program work correctly. By including the preceding cases in our switch statement, we prevent the error message in the default case from being printed each time a newline, tab or space is encountered in the input.

Listing several case labels together (such as case 'D': case 'd': in Fig. 4.7) simply means that the same set of actions is to occur for either of these cases.

When using the switch statement, remember that each individual case can test only a constant integral expression—i.e., any combination of character constants and integer constants that evaluates to a constant integer value. A character constant is represented as the specific character in single quotes, such as 'A'. Characters must be enclosed within single quotes to be recognized as character constants—characters in double quotes are recognized as strings. Integer constants are simply integer values. In our example, we have used character constants. Remember that characters are represented as small integer values.
Notes on Integral Types
Portable languages like C must have flexible data type sizes. Different applications may need integers of different sizes. C provides several data types to represent integers. The range of values for each type depends on the particular computer’s hardware. In addition to int and char, C provides types short (an abbreviation of short int) and long (an abbreviation of long int). C specifies that the minimum range of values for short integers is –32768 to +32767. For the vast majority of integer calculations, long integers are sufficient. The standard specifies that the minimum range of values for long integers is –2147483648 to +2147483647. The standard states that the range of values for an int is at least the same as the range for short integers and no larger than the range for long integers. The data type signed char can be used to represent integers in the range –128 to +127 or any of the characters in the computer’s character set.

4.8 do...while Repetition Statement

The do...while repetition statement is similar to the while statement. In the while statement, the loop-continuation condition is tested at the beginning of the loop before the body of the loop is performed. The do...while statement tests the loop-continuation condition after the loop body is performed. Therefore, the loop body will be executed at least once. When a do...while terminates, execution continues with the statement after the while clause. It’s not necessary to use braces in the do...while statement if there is only one statement in the body. However, the braces are usually included to avoid confusion between the while and do...while statements. For example,

```c
while ( condition )
```

is normally regarded as the header to a while statement. A do...while with no braces around the single-statement body appears as

```c
do
    statement
while ( condition );
```

which can be confusing. The last line—while( condition );—may be misinterpreted by as a while statement containing an empty statement. Thus, to avoid confusion, the do...while with one statement is often written as follows:

```c
do {
    statement
} while ( condition );
```

Good Programming Practice 4.8
To eliminate the potential for ambiguity, some programmers always include braces in a do...while statement, even if the braces are not necessary.

Common Programming Error 4.7
Infinite loops are caused when the loop-continuation condition in a while, for or do...while statement never becomes false. To prevent this, make sure there is not a semicolon immediately after the header of a while or for statement. In a counter-controlled loop, make sure the control variable is incremented (or decremented) in the loop. In a sentinel-controlled loop, make sure the sentinel value is eventually input.
Figure 4.9 uses a do...while statement to print the numbers from 1 to 10. The control variable counter is preincremented in the loop-continuation test. Note also the use of the braces to enclose the single-statement body of the do...while.

```c
/* Fig. 4.9: fig04_09.c */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    int counter = 1; /* initialize counter */
    do {
        printf( "%d  ", counter ); /* display counter */
    } while ( ++counter <= 10 ); /* end do...while */
    return 0; /* indicate program ended successfully */
} /* end function main */
```

Figure 4.10 shows the do...while statement flowchart, which makes it clear that the loop-continuation condition does not execute until after the action is performed at least once.

![Flowcharting the do...while repetition statement.](image)

### 4.9 break and continue Statements

The break and continue statements are used to alter the flow of control. The break statement, when executed in a while, for, do...while or switch statement, causes an immediate exit from that statement. Program execution continues with the next statement. Common uses of the break statement are to escape early from a loop or to skip the remainder of a switch statement (as in Fig. 4.7). Figure 4.11 demonstrates the break statement.
in a for repetition statement. When the if statement detects that \( x \) has become 5, break is executed. This terminates the for statement, and the program continues with the printf after the for. The loop fully executes only four times.

```c
/* Fig. 4.11: fig04_11.c */
#include <stdio.h>

int main()
{
    int x; /* counter */
    /* loop 10 times */
    for ( x = 1; x <= 10; x++ )
    {
        /* if x is 5, terminate loop */
        if ( x == 5 )
        {
            break; /* break loop only if x is 5 */
        } /* end if */
        printf("%d ", x); /* display value of x */
    } /* end for */
    printf("\nBroke out of loop at x == %d\n", x);
    return 0; /* indicate program ended successfully */
} /* end function main */
```

**Fig. 4.11** | Using the break statement in a for statement.

The continue statement, when executed in a while, for or do...while statement, skips the remaining statements in the body of that control statement and performs the next iteration of the loop. In while and do...while statements, the loop-continuation test is evaluated immediately after the continue statement is executed. In the for statement, the increment expression is executed, then the loop-continuation test is evaluated. Earlier, we said that the while statement could be used in most cases to represent the for statement. The one exception occurs when the increment expression in the while statement follows the continue statement. In this case, the increment is not executed before the repetition-continuation condition is tested, and the while does not execute in the same manner as the for. Figure 4.12 uses the continue statement in a for statement to skip the printf statement and begin the next iteration of the loop.

```c
/* Fig. 4.12: fig04_12.c */
#include <stdio.h>

int main()
{
    int x; /* counter */
    /* loop 10 times */
    for ( x = 1; x <= 10; x++ )
    {
        /* if x is 5, terminate loop */
        if ( x == 5 )
        {
            continue; /* continue only if x is 5 */
        } /* end if */
        printf("%d ", x); /* display value of x */
    } /* end for */
    printf("\nContinued out of loop at x == %d\n", x);
    return 0; /* indicate program ended successfully */
} /* end function main */
```

**Fig. 4.12** | Using the continue statement in a for statement. (Part 1 of 2.)
Chapter 4  C Program Control

4.10 Logical Operators

So far we have studied only simple conditions, such as counter <= 10, total > 1000, and number != sentinelValue. We’ve expressed these conditions in terms of the relational operators, >, <, >= and <=, and the equality operators, == and !=. Each decision tested precisely one condition. To test multiple conditions in the process of making a decision, we had to perform these tests in separate statements or in nested if or if...else statements.

C provides logical operators that may be used to form more complex conditions by combining simple conditions. The logical operators are && (logical AND), || (logical

```c
/* function main begins program execution */
int main( void )
{
    int x; /* counter */
    /* loop 10 times */
    for ( x = 1; x <= 10; x++ ) {
        /* if x is 5, continue with next iteration of loop */
        if ( x == 5 ) {
            continue; /* skip remaining code in loop body */
        } /* end if */
        printf( "%d ", x ); /* display value of x */
    } /* end for */
    printf( "\nUsed continue to skip printing the value 5\n" );
    return 0; /* indicate program ended successfully */
} /* end function main */
```

Fig. 4.12  |  Using the continue statement in a for statement. (Part 2 of 2.)

Software Engineering Observation 4.2

Some programmers feel that break and continue violate the norms of structured programming. The effects of these statements can be achieved by structured programming techniques we'll soon learn, so these programmers do not use break and continue.

Performance Tip 4.1

The break and continue statements, when used properly, perform faster than the corresponding structured techniques that we'll soon learn.

Software Engineering Observation 4.3

There is a tension between achieving quality software engineering and achieving the best-performing software. Often one of these goals is achieved at the expense of the other.

4.10 Logical Operators

So far we have studied only simple conditions, such as counter <= 10, total > 1000, and number != sentinelValue. We’ve expressed these conditions in terms of the relational operators, >, <, >= and <=, and the equality operators, == and !=. Each decision tested precisely one condition. To test multiple conditions in the process of making a decision, we had to perform these tests in separate statements or in nested if or if...else statements.

C provides logical operators that may be used to form more complex conditions by combining simple conditions. The logical operators are && (logical AND), || (logical
OR) and ! (logical NOT also called logical negation). We’ll consider examples of each of these operators.

Suppose we wish to ensure that two conditions are both true before we choose a certain path of execution. In this case, we can use the logical operator && as follows:

```c
if ( gender == 1 && age >= 65 )
    ++seniorFemales;
```

This if statement contains two simple conditions. The condition `gender == 1` might be evaluated, for example, to determine if a person is a female. The condition `age >= 65` is evaluated to determine if a person is a senior citizen. The two simple conditions are evaluated first because the precedences of `==` and `>=` are both higher than the precedence of `&&`. The if statement then considers the combined condition

```c
gender == 1 && age >= 65
```

This condition is true if and only if both of the simple conditions are true. Finally, if this combined condition is indeed true, then the count of `seniorFemales` is incremented by 1. If either or both of the simple conditions are false, then the program skips the incrementing and proceeds to the statement following the if.

Figure 4.13 summarizes the `&&` operator. The table shows all four possible combinations of zero (false) and nonzero (true) values for `expression1` and `expression2`. Such tables are often called truth tables. C evaluates all expressions that include relational operators, equality operators, and/or logical operators to 0 or 1. Although C sets a true value to 1, it accepts any nonzero value as true.

<table>
<thead>
<tr>
<th><code>expression1</code></th>
<th><code>expression2</code></th>
<th><code>expression1 &amp;&amp; expression2</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>0</td>
</tr>
<tr>
<td>nonzero</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nonzero</td>
<td>nonzero</td>
<td>1</td>
</tr>
</tbody>
</table>

![Fig. 4.13](image)

Now let’s consider the `||` (logical OR) operator. Suppose we wish to ensure at some point in a program that either or both of two conditions are true before we choose a certain path of execution. In this case, we use the `||` operator as in the following program segment:

```c
if ( semesterAverage >= 90 || finalExam >= 90 )
    printf("Student grade is A\n");
```

This statement also contains two simple conditions. The condition `semesterAverage >= 90` is evaluated to determine if the student deserves an “A” in the course because of a solid performance throughout the semester. The condition `finalExam >= 90` is evaluated to determine if the student deserves an “A” in the course because of an outstanding performance on the final exam. The if statement then considers the combined condition

```c
semesterAverage >= 90 || finalExam >= 90
```
and awards the student an “A” if either or both of the simple conditions are true. The message “Student grade is A” is not printed only when both of the simple conditions are false (zero). Figure 4.14 is a truth table for the logical OR operator (||).

| expression1 | expression2 | expression1 || expression2 |
|-------------|-------------|-------------|
| 0           | 0           | 0           |
| 0           | nonzero     | 1           |
| nonzero     | 0           | 1           |
| nonzero     | nonzero     | 1           |

**Fig. 4.14 | Truth table for the logical OR (||) operator.**

The && operator has a higher precedence than ||. Both operators associate from left to right. An expression containing && or || operators is evaluated only until truth or falsehood is known. Thus, evaluation of the condition

```
gender == 1 && age >= 65
```

will stop if gender is not equal to 1 (i.e., the entire expression is false), and continue if gender is equal to 1 (i.e., the entire expression could still be true if age >= 65). This performance feature for the evaluation of logical AND and logical OR expressions is called **short-circuit evaluation**.

C provides ! (logical negation) to enable a programmer to “reverse” the meaning of a condition. Unlike operators && and ||, which combine two conditions (and are therefore binary operators), the logical negation operator has only a single condition as an operand (and is therefore a unary operator). The logical negation operator is placed before a condition when we’re interested in choosing a path of execution if the original condition (without the logical negation operator) is false, such as in the following program segment:

```
if ( !( grade == sentinelValue ) )
  printf( "The next grade is %f\n", grade );
```

The parentheses around the condition grade == sentinelValue are needed because the logical negation operator has a higher precedence than the equality operator. Figure 4.15 is a truth table for the logical negation operator.

<table>
<thead>
<tr>
<th>expression</th>
<th>!expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nonzero</td>
<td>0</td>
</tr>
</tbody>
</table>

**Fig. 4.15 | Truth table for operator ! (logical negation).**
In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational operator. For example, the preceding statement may also be written as follows:

```c
if ( grade != sentinelValue )
    printf( "The next grade is %f\n", grade );
```

Figure 4.16 shows the precedence and associativity of the operators introduced to this point. The operators are shown from top to bottom in decreasing order of precedence.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>++ (postfix)</td>
<td>right to left</td>
<td>postfix</td>
</tr>
<tr>
<td>-- (postfix)</td>
<td>right to left</td>
<td>postfix</td>
</tr>
<tr>
<td>+ - !</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>++ (prefix) -- (prefix) (type)</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
<td>conditional</td>
</tr>
<tr>
<td>= += -= *= /= %=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
<td>comma</td>
</tr>
</tbody>
</table>

Fig. 4.16 | Operator precedence and associativity.

4.11 Confusing Equality (==) and Assignment (=) Operators

There is one type of error that C programmers, no matter how experienced, tend to make so frequently that we felt it was worth a separate section. That error is accidentally swapping the operators `==` (equality) and `=` (assignment). What makes these swaps so damaging is the fact that they do not ordinarily cause compilation errors. Rather, statements with these errors ordinarily compile correctly, allowing programs to run to completion while likely generating incorrect results through runtime logic errors.

Two aspects of C cause these problems. One is that any expression in C that produces a value can be used in the decision portion of any control statement. If the value is 0, it’s treated as false, and if the value is nonzero, it’s treated as true. The second is that assignments in C produce a value, namely the value that is assigned to the variable on the left side of the assignment operator. For example, suppose we intend to write

```c
if ( payCode == 4 )
    printf( "You get a bonus!" );
```

but we accidentally write

```c
if ( payCode = 4 )
    printf( "You get a bonus!" );
```
The first if statement properly awards a bonus to the person whose paycode is equal to 4. The second if statement—the one with the error—evaluates the assignment expression in the if condition. This expression is a simple assignment whose value is the constant 4. Because any nonzero value is interpreted as “true,” the condition in this if statement is always true, and not only is the value of payCode inadvertently set to 4, but the person always receives a bonus regardless of what the actual paycode is!

**Common Programming Error 4.8**

Using operator == for assignment or using operator = for equality is a logic error.

Programmers normally write conditions such as \( x == 7 \) with the variable name on the left and the constant on the right. By reversing these terms so that the constant is on the left and the variable name is on the right, as in \( 7 == x \), the programmer who accidentally replaces the == operator with = is protected by the compiler. The compiler will treat this as a syntax error, because only a variable name can be placed on the left-hand side of an assignment expression. At least this will prevent the potential devastation of a runtime logic error.

Variable names are said to be *lvalues* (for “left values”) because they can be used on the left side of an assignment operator. Constants are said to be *rvalues* (for “right values”) because they can be used on only the right side of an assignment operator. *Lvalues* can also be used as *rvalues*, but not vice versa.

**Good Programming Practice 4.9**

When an equality expression has a variable and a constant, as in \( x == 1 \), some programmers prefer to write the expression with the constant on the left and the variable name on the right (e.g. \( 1 == x \)) as protection against the logic error that occurs when you accidentally replace operator == with =).

The other side of the coin can be equally unpleasant. Suppose you want to assign a value to a variable with a simple statement like

\[
x = 1;
\]

but instead write

\[
x == 1;
\]

Here, too, this is not a syntax error. Rather the compiler simply evaluates the conditional expression. If \( x \) is equal to 1, the condition is true and the expression returns the value 1. If \( x \) is not equal to 1, the condition is false and the expression returns the value 0. Regardless of what value is returned, there is no assignment operator, so the value is simply lost, and the value of \( x \) remains unaltered, probably causing an execution-time logic error. Unfortunately, we do not have a handy trick available to help you with this problem! Many compilers, however, will issue a warning on such a statement.

**Error-Prevention Tip 4.6**

After you write a program, text search it for every = and check that it’s used properly.
4.12 Structured Programming Summary

Just as architects design buildings by employing the collective wisdom of their profession, so should programmers design programs. Our field is younger than architecture is, and our collective wisdom is considerably sparser. We have learned a great deal in a mere six decades. Perhaps most important, we have learned that structured programming produces programs that are easier (than unstructured programs) to understand and hence are easier to test, debug, modify, and even prove correct in a mathematical sense.

Chapters 3 and 4 have concentrated on C’s control statements. Each statement has been presented, flowcharted and discussed separately with examples. Now, we summarize the results of Chapters 3 and 4 and introduce a simple set of rules for the formation and properties of structured programs.

Figure 4.17 summarizes the control statements discussed in Chapters 3 and 4. Small circles are used in the figure to indicate the single entry point and the single exit point of each statement. Connecting individual flowchart symbols arbitrarily can lead to unstructured programs. Therefore, the programming profession has chosen to combine flowchart symbols to form a limited set of control statements, and to build only structured programs by properly combining control statements in two simple ways. For simplicity, only single-entry/single-exit control statements are used—there is only one way to enter and only one way to exit each control statement. Connecting control statements in sequence to form structured programs is simple—the exit point of one control statement is connected directly to the entry point of the next, i.e., the control statements are simply placed one after another in a program—we have called this “control-statement stacking.” The rules for forming structured programs also allow for control statements to be nested.

Figure 4.18 shows the rules for forming structured programs. The rules assume that the rectangle flowchart symbol may be used to indicate any action including input/output. Figure 4.19 shows the simplest flowchart.

Applying the rules of Fig. 4.18 always results in a structured flowchart with a neat, building-block appearance. Repeatedly applying Rule 2 to the simplest flowchart (Fig. 4.19) results in a structured flowchart containing many rectangles in sequence (Fig. 4.20). Notice that Rule 2 generates a stack of control statements; so we call Rule 2 the stacking rule.

Rule 3 is called the nesting rule. Repeatedly applying Rule 3 to the simplest flowchart results in a flowchart with neatly nested control statements. For example, in Fig. 4.21, the rectangle in the simplest flowchart is first replaced with a double-selection (if...else) statement. Then Rule 3 is applied again to both of the rectangles in the double-selection statement, replacing each of these rectangles with double-selection statements. The dashed box around each of the double-selection statements represents the rectangle that was replaced in the original flowchart.

Rule 4 generates larger, more involved, and more deeply nested structures. The flowcharts that emerge from applying the rules in Fig. 4.18 constitute the set of all possible structured flowcharts and hence the set of all possible structured programs.

It’s because of the elimination of the goto statement that these building blocks never overlap one another. The beauty of the structured approach is that we use only a small number of simple single-entry/single-exit pieces, and we assemble them in only two simple ways. Figure 4.22 shows the kinds of stacked building blocks that emerge from applying Rule 2 and the kinds of nested building blocks that emerge from applying Rule 3. The
Fig. 4.17 | C’s single-entry/single-exit sequence, selection and repetition statements.
figure also shows the kind of overlapped building blocks that cannot appear in structured flowcharts (because of the elimination of the goto statement).

### Rules for Forming Structured Programs

1) Begin with the “simplest flowchart” (Fig. 4.19).
2) Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
3) Any rectangle (action) can be replaced by any control statement (sequence, if, if…else, switch, while, do…while or for).
4) Rules 2 and 3 may be applied as often as you like and in any order.

**Fig. 4.18**  |  Rules for forming structured programs.

![Diagram](image1)

**Fig. 4.19**  |  Simplest flowchart.

![Diagram](image2)

**Fig. 4.20**  |  Repeatedly applying Rule 2 of Fig. 4.18 to the simplest flowchart.
If the rules in Fig. 4.18 are followed, an unstructured flowchart (such as that in Fig. 4.23) cannot be created. If you’re uncertain whether a particular flowchart is structured, apply the rules of Fig. 4.18 in reverse to try to reduce the flowchart to the simplest flowchart. If you succeed, the original flowchart is structured; otherwise, it’s not.

Structured programming promotes simplicity. Bohm and Jacopini showed that only three forms of control are needed:

- Sequence
- Selection
- Repetition
Sequence is straightforward. Selection is implemented in one of three ways:

- if statement (single selection)
- if...else statement (double selection)
- switch statement (multiple selection)

In fact, it’s straightforward to prove that the simple if statement is sufficient to provide any form of selection—everything that can be done with the if...else statement and the switch statement can be implemented with one or more if statements.

Repetition is implemented in one of three ways:

- while statement
• `do...while` statement
• `for` statement

It’s straightforward to prove that the `while` statement is sufficient to provide any form of repetition. Everything that can be done with the `do...while` statement and the `for` statement can be done with the `while` statement.

Combining these results illustrates that any form of control ever needed in a C program can be expressed in terms of only three forms of control:

• sequence
• `if` statement (selection)
• `while` statement (repetition)

And these control statements can be combined in only two ways—stacking and nesting. Indeed, structured programming promotes simplicity.

In Chapters 3 and 4, we discussed how to compose programs from control statements containing actions and decisions. In Chapter 5, we introduce another program structuring unit called the function. We’ll learn to compose large programs by combining functions, which, in turn, are composed of control statements. We’ll also discuss how using functions promotes software reusability.

**Summary**

**Section 4.2 Repetition Essentials**

• Most programs involve repetition, or looping. A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true.

• Counter-controlled repetition is sometimes called definite repetition because we know in advance exactly how many times the loop will execute.

• Sentinel-controlled repetition is sometimes called indefinite repetition because it’s not known in advance how many times the loop will execute.

• In counter-controlled repetition, a control variable is used to count the number of repetitions. The control variable is incremented (usually by 1) each time the group of instructions is performed. When the correct number of repetitions has been performed, the loop terminates, and the program resumes execution with the statement after the repetition statement.

• Sentinel values are used to control repetition when the number of repetitions is not known in advance, and the loop includes statements that obtain data each time the loop is performed.

• The sentinel value indicates “end of data.” The sentinel is entered after all regular data items have been supplied to the program. Sentinels must be distinct from regular data items.

**Section 4.3 Counter-Controlled Repetition**

• Counter-controlled repetition requires the name of a control variable (or loop counter), the initial value of the control variable, the increment (or decrement) by which the control variable is modified each time through the loop, and the condition that tests for the final value of the control variable (i.e., whether looping should continue).

**Section 4.4 `for` Repetition Statement**

• The `for` repetition statement handles all the details of counter-controlled repetition.
When the `for` statement begins executing, its control variable is initialized. Then, the loop-continuation condition is checked. If the condition is true, the loop’s body executes. The control variable is then incremented, and the loop begins again with the loop-continuation condition. This process continues until the loop-continuation condition fails.

The general format of the `for` statement is

```c
for ( expression1; expression2; expression3 )
statement
```

where `expression1` initializes the loop-control variable, `expression2` is the loop-continuation condition, and `expression3` increments the control variable.

In most cases, the `for` statement can be represented with an equivalent `while` statement as in:

```c
expression1;
while ( expression2 ) {
    statement
    expression3;
}
```

The comma operator guarantees that lists of expressions evaluate from left to right. The value of the entire expression is that of the rightmost expression.

The three expressions in the `for` statement are optional. If `expression2` is omitted, C assumes that the condition is true, thus creating an infinite loop. One might omit `expression1` if the control variable is initialized elsewhere in the program. `expression3` might be omitted if the increment is calculated by statements in the body of the `for` statement or if no increment is needed.

The increment expression in the `for` statement acts like a stand-alone C statement at the end of the body of the `for`.

The two semicolons in the `for` statement are required.

### Section 4.5 for Statement: Notes and Observations

- The initialization, loop-continuation condition and increment can contain arithmetic expressions.
- The “increment” may be negative (in which case it’s really a decrement and the loop actually counts downward).
- If the loop-continuation condition is initially false, the body portion of the loop is not performed. Instead, execution proceeds with the statement following the `for` statement.

### Section 4.6 Examples Using the for Statement

- Function `pow` performs exponentiation. The function `pow(x, y)` calculates the value of `x` raised to the `y`th power. It takes two arguments of type `double` and returns a `double` value.
- Type `double` is a floating-point type much like `float`, but typically a variable of type `double` can store a value of much greater magnitude with greater precision than `float`.
- The header `<math.h>` should be included whenever a math function such as `pow` is used.
- The conversion specifier `%21.2f` denotes that a floating-point value will be displayed right justified in a field of 21 characters with two digits to the right of the decimal point.
- To left justify a value in a field, place a `-` (minus sign) between the `%` and the field width.

### Section 4.7 switch Multiple-Selection Statement

- Occasionally, an algorithm will contain a series of decisions in which a variable or expression is tested separately for each of the constant integral values it may assume, and different actions are taken. This is called multiple selection. C provides the `switch` statement to handle this.
- The `switch` statement consists of a series of `case` labels, an optional `default` case and statements to execute for each case.
• The `getchar` function (from the standard input/output library) reads and returns one character from the keyboard.
• Characters are normally stored in variables of type `char`. Characters can be stored in any integer data type because they’re usually represented as one-byte integers in the computer. Thus, we can treat a character as either an integer or a character, depending on its use.
• Many computers today use the ASCII (American Standard Code for Information Interchange) character set in which 97 represents the lowercase letter ‘a’.
• Characters can be read with `scanf` by using the conversion specifier `%c`.
• Assignment expressions as a whole actually have a value. This value is assigned to the variable on the left side of the `=`.
• The fact that assignment statements have values can be useful for setting several variables to the same value, as in `a = b = c = 0;`.
• `EOF` is often used as a sentinel value. `EOF` is a symbolic integer constant defined in `<stdio.h>`.
• On Linux/UNIX systems and many others, the `EOF` indicator is entered by typing `<Ctrl> d`. On other systems, such as Microsoft Windows, the `EOF` indicator can be entered by typing `<Ctrl> z`.
• Keyword `switch` is followed by the controlling expression in parentheses. The value of this expression is compared with each of the case labels. If a match occurs, the statements for that case execute. If no match occurs, the `default` case executes.
• The `break` statement causes program control to continue with the statement after the `switch`. The `break` statement prevents the `case` s in a `switch` statement from running together.
• Each `case` can have one or more actions. The `switch` statement is different from all other control statements in that braces are not required around multiple actions in a case of a `switch`.
• Listing several `case` labels together simply means that the same set of actions is to occur for any of these cases.
• Remember that the `switch` statement can be used only for testing a constant integral expression—i.e., any combination of character constants and integer constants that evaluates to a constant integer value. A character constant is represented as the specific character in single quotes, such as ‘A’. Characters must be enclosed within single quotes to be recognized as character constants. Integer constants are simply integer values.
• C provides several data types to represent integers. The range of integer values for each type depends on the particular computer’s hardware. In addition to the types `int` and `char`, C provides types `short` (an abbreviation of `short int`) and `long` (an abbreviation of `long int`). The minimum range of values for `short` integers is −32768 to +32767. For the vast majority of integer calculations, `long` integers are sufficient. The standard specifies that the minimum range of values for `long` integers is −2147483648 to +2147483647. The standard states that the range of values for an `int` is at least the same as the range for `short` integers and no larger than the range for `long` integers. The data type `signed char` can be used to represent integers in the range −128 to +127 or any of the characters in the computer’s character set.

Section 4.8 do…while Repetition Statement
• The `do…while` statement tests the loop-continuation condition after the loop body is performed. Therefore, the loop body will be executed at least once. When a `do…while` terminates, execution continues with the statement after the `while` clause.

Section 4.9 break and continue Statements
• The `break` statement, when executed in a `while`, `for`, `do…while` or `switch` statement, causes immediate exit from that statement. Program execution continues with the next statement.
• The `continue` statement, when executed in a `while`, `for` or `do...while` statement, skips the remaining statements in the body of that control statement and performs the next iteration of the loop. In `while` and `do...while` statements, the loop-continuation test is evaluated immediately after the `continue` statement is executed. In the `for` statement, the increment expression is executed, then the loop-continuation test is evaluated.

Section 4.10 Logical Operators

• Logical operators may be used to form complex conditions by combining simple conditions. The logical operators are `&&` (logical AND), `||` (logical OR) and `!` (logical NOT, or logical negation).
• A condition containing the `&&` (logical AND) operator is true if and only if both of the simple conditions are true.
• C evaluates all expressions that include relational operators, equality operators, and/or logical operators to 0 or 1. Although C sets a true value to 1, it accepts any nonzero value as true.
• A condition containing the `||` (logical OR) operator is true if either or both of the simple conditions are true.
• The `&&` operator has a higher precedence than `||`. Both operators associate from left to right.
• An expression containing `&&` or `||` operators is evaluated only until truth or falsehood is known.
• C provides `!` (logical negation) to enable a programmer to “reverse” the meaning of a condition. Unlike the binary operators `&&` and `||`, which combine two conditions, the unary logical negation operator has only a single condition as an operand.
• The logical negation operator is placed before a condition when we’re interested in choosing a path of execution if the original condition (without the logical negation operator) is false.
• In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational operator.

Section 4.11 Confusing Equality (`==`) and Assignment (`=`) Operators

• Programmers often accidentally swap the operators `==` (equality) and `=` (assignment). What makes these swaps so damaging is that they do not ordinarily cause syntax errors. Rather, statements with these errors ordinarily compile correctly, allowing programs to run to completion while likely generating incorrect results through runtime logic errors.
• Programmers normally write conditions such as `x == 7` with the variable name on the left and the constant on the right. By reversing these terms so that the constant is on the left and the variable name is on the right, as in `7 == x`, the programmer who accidentally replaces the `==` operator with `=` will be protected by the compiler. The compiler will treat this as a syntax error, because only a variable name can be placed on the left-hand side of an assignment statement.
• Variable names are said to be `lvalues` (for “left values”) because they can be used on the left side of an assignment operator.
• Constants are said to be `rvalues` (for “right values”) because they can be used only on the right side of an assignment operator. `lvalues` can also be used as `rvalues`, but not vice versa.

Terminology

ASCII (American Standard Code for Information Interchange) character set 110
	case label 111
char primitive type 109
comma operator 102
constant integral expression 112
control variable 98
controlling expression in a `switch` 111
decrement a control variable 99
definite repetition 98
final value of a control variable 99
function prototype 107
increment a control variable 99
indefinite repetition 98
initial value of a control variable 99
logical AND operator (&&) 116
logical negation operator (!) 117
logical OR operator (||) 117
loop-continuation condition 98
lvalue (“left value”) 120
name of a control variable 99
nesting rule 121
off-by-one error 101
pow (power) function 107
rvalue (“right value”) 120
short-circuit evaluation 118
stacking rule 121
truth table 117

Self-Review Exercises

4.1 Fill in the blanks in each of the following statements.
   a) Counter-controlled repetition is also known as _______ repetition because it’s known in advance how many times the loop will be executed.
   b) Sentinel-controlled repetition is also known as _______ repetition because it’s not known in advance how many times the loop will be executed.
   c) In counter-controlled repetition, a(n) _______ is used to count the number of times a group of instructions should be repeated.
   d) The _______ statement, when executed in a repetition statement, causes the next iteration of the loop to be performed immediately.
   e) The _______ statement, when executed in a repetition statement or a switch, causes an immediate exit from the statement.
   f) The _______ is used to test a particular variable or expression for each of the constant integral values it may assume.

4.2 State whether the following are true or false. If the answer is false, explain why.
   a) The default case is required in the switch selection statement.
   b) The break statement is required in the default case of a switch selection statement.
   c) The expression (x > y && a < b) is true if either x > y is true or a < b is true.
   d) An expression containing the || operator is true if either or both of its operands is true.

4.3 Write a statement or a set of statements to accomplish each of the following tasks:
   a) Sum the odd integers between 1 and 99 using a for statement. Assume the integer variables sum and count have been defined.
   b) Print the value 333.546372 in a field width of 15 characters with precisions of 1, 2, 3, 4 and 5. Left justify the output. What are the five values that print?
   c) Calculate the value of 2.5 raised to the power of 3 using the pow function. Print the result with a precision of 2 in a field width of 10 positions. What is the value that prints?
   d) Print the integers from 1 to 20 using a while loop and the counter variable x. Assume that the variable x has been defined, but not initialized. Print only five integers per line.
   [Hint: Use the calculation x % 5. When the value of this is 0, print a newline character, otherwise print a tab character.]
   e) Repeat Exercise 4.3 (d) using a for statement.

4.4 Find the error in each of the following code segments and explain how to correct it.
   a) x = 1;
      while ( x <= 10 );
      x++;
   }
   b) for ( y = .1; y != 1.0; y += .1 )
      printf( "%.f\n", y );
c) `switch ( n ) {
  case 1:
    printf( "The number is 1\n" );
  case 2:
    printf( "The number is 2\n" );
    break;
  default:
    printf( "The number is not 1 or 2\n" );
    break;
}

d) The following code should print the values 1 to 10.

```c
n = 1;
while ( n < 10 )
  printf( "%d \n", n++ );
```

### Answers to Self-Review Exercises

**4.1**

a) definite. b) indefinite. c) control variable or counter. d) continue. e) break. f) switch selection statement.

**4.2**

a) False. The `default` case is optional. If no default action is needed, then there is no need for a `default` case.

b) False. The `break` statement is used to exit the `switch` statement. The `break` statement is not required when the `default` case is the last case.

c) False. Both of the relational expressions must be true in order for the entire expression to be true when using the `&&` operator.

d) True.

**4.3**

a) `sum = 0;
   for ( count = 1; count <= 99; count += 2 ) {
   sum += count;
   }
   printf( "%-15.1f\n", 333.546372 ); /* prints 333.5 */
   printf( "%-15.2f\n", 333.546372 ); /* prints 333.55 */
   printf( "%-15.3f\n", 333.546372 ); /* prints 333.546 */
   printf( "%-15.4f\n", 333.546372 ); /* prints 333.5464 */
   printf( "%-15.5f\n", 333.546372 ); /* prints 333.54637 */
   printf( "%10.2f\n", pow( 2.5, 3 ) ); /* prints 15.63 */
   }

b) `x = 1;
   while ( x <= 20 ) {
   printf( "%d", x );
   if ( x % 5 == 0 ) {
   printf( "\n" );
   }
   else {
   printf( "\t" );
   }
   x++;
   }
   ` or

or
4.4 a) Error: The semicolon after the while header causes an infinite loop. 
Correction: Replace the semicolon with a { or remove both the ; and the }.

b) Error: Using a floating-point number to control a for repetition statement. 
Correction: Use an integer, and perform the proper calculation to get the values you desire.

```c
for ( y = 1; y != 10; y++ )
    printf( "%f\n", ( float ) y / 10 );
```

c) Error: Missing break statement in the statements for the first case.
Correction: Add a break statement at the end of the statements for the first case. This is not necessarily an error if you want the statement of case 2: to execute every time the case 1: statement executes.

d) Error: Improper relational operator used in the while repetition-continuation condition.
Correction: Use <= rather than <.

Exercises

4.5 Find the error in each of the following. (Note: There may be more than one error.)
a) For ( x = 100, x >= 1, x++ )
   printf( "%d\n", x );
b) The following code should print whether a given integer is odd or even:
   switch ( value % 2 ) {
     case 0:
       printf( "Even integer\n" );
     case 1:
       printf( "Odd integer\n" );
   }
c) The following code should input an integer and a character and print them. Assume the user types as input 100 A.
   scanf( "%d", &intVal );
   charVal = getchar();
   printf( "Integer: %d
Character: %c
", intVal, charVal );
d) for ( x = .000001; x == .0001; x += .000001 ) {
   printf( "%d\n", x );
}
e) The following code should output the odd integers from 999 to 1:
   for ( x = 999; x >= 1; x += 2 ) {
   printf( "%d\n", x );
}
f) The following code should output the even integers from 2 to 100:
   counter = 2;
   Do {
     if ( counter % 2 == 0 ) {
       printf( "%d\n", counter );
     }
     counter += 2;
   } While ( counter < 100 );
g) The following code should sum the integers from 100 to 150 (assume total is initialized to 0):
   for ( x = 100; x <= 150; x++ ) {
   total += x;
}

4.6 State which values of the control variable x are printed by each of the following for statements:
a) for ( x = 2; x <= 13; x += 2 ) {
   printf( "%d\n", x );
}
Chapter 4  C Program Control

b) for ( x = 5; x <= 22; x += 7 ) {
    printf( "%d\n", x );
}
c) for ( x = 3; x <= 15; x += 3 ) {
    printf( "%d\n", x );
}
d) for ( x = 1; x <= 5; x += 7 ) {
    printf( "%d\n", x );
}
e) for ( x = 12; x >= 2; x -= 3 ) {
    printf( "%d\n", x );
}

4.7 Write for statements that print the following sequences of values:
a) 1, 2, 3, 4, 5, 6, 7
b) 3, 8, 13, 18, 23
c) 20, 14, 8, 2, –4, –10
d) 19, 27, 35, 43, 51

4.8 What does the following program do?

```
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    int x;
    int y;
    int i;
    int j;

    /* prompt user for input */
    printf( "Enter two integers in the range 1-20: " );
    scanf( "%d%d", &x, &y ); /* read values for x and y */

    for ( i = 1; i <= y; i++ ) { /* count from 1 to y */
        for ( j = 1; j <= x; j++ ) { /* count from 1 to x */
            printf( "@" ); /* output @ */
        } /* end inner for */
    } /* end outer for */
    printf( "\n" ); /* begin new line */
    return 0; /* indicate program ended successfully */
} /* end function main */
```

4.9 (Sum a Sequence of Integers) Write a program that sums a sequence of integers. Assume that the first integer read with scanf specifies the number of values remaining to be entered. Your program should read only one value each time scanf is executed. A typical input sequence might be

```
5 100 200 300 400 500
```

where the 5 indicates that the subsequent five values are to be summed.

4.10 (Average a Sequence of Integers) Write a program that calculates and prints the average of several integers. Assume the last value read with scanf is the sentinel 9999. A typical input sequence might be
indicating that the average of all the values preceding 9999 is to be calculated.

4.11  *(Find the Smallest)*  Write a program that finds the smallest of several integers. Assume that the first value read specifies the number of values remaining.

4.12  *(Calculating the Sum of Even Integers)*  Write a program that calculates and prints the sum of the even integers from 2 to 30.

4.13  *(Calculating the Product of Odd Integers)*  Write a program that calculates and prints the product of the odd integers from 1 to 15.

4.14  *(Factorials)*  The factorial function is used frequently in probability problems. The factorial of a positive integer \( n \) (written \( n! \) and pronounced “\( n \) factorial”) is equal to the product of the positive integers from 1 to \( n \). Write a program that evaluates the factorials of the integers from 1 to 5. Print the results in tabular format. What difficulty might prevent you from calculating the factorial of 20?

4.15  *(Modified Compound Interest Program)* Modify the compound-interest program of Section 4.6 to repeat its steps for interest rates of 5%, 6%, 7%, 8%, 9%, and 10%. Use a for loop to vary the interest rate.

4.16  *(Triangle Printing Program)* Write a program that prints the following patterns separately, one below the other. Use for loops to generate the patterns. All asterisks (*) should be printed by a single printf statement of the form printf("*"); (this causes the asterisks to print side by side). [Hint: The last two patterns require that each line begin with an appropriate number of blanks.]

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*******</td>
<td>**********</td>
<td>*</td>
</tr>
<tr>
<td>**</td>
<td>*******</td>
<td>**********</td>
<td>**</td>
</tr>
<tr>
<td>***</td>
<td>*******</td>
<td>**********</td>
<td>***</td>
</tr>
<tr>
<td>****</td>
<td>*******</td>
<td>**********</td>
<td>****</td>
</tr>
<tr>
<td>*****</td>
<td>*******</td>
<td>**********</td>
<td>*****</td>
</tr>
<tr>
<td>******</td>
<td>*******</td>
<td>**********</td>
<td>******</td>
</tr>
<tr>
<td>*******</td>
<td>*******</td>
<td>**********</td>
<td>*******</td>
</tr>
<tr>
<td>********</td>
<td>*******</td>
<td>**********</td>
<td>********</td>
</tr>
<tr>
<td>*********</td>
<td>*******</td>
<td>**********</td>
<td>*********</td>
</tr>
<tr>
<td>**********</td>
<td>*******</td>
<td>**********</td>
<td>**********</td>
</tr>
</tbody>
</table>

4.17  *(Calculating Credit Limits)* Collecting money becomes increasingly difficult during periods of recession, so companies may tighten their credit limits to prevent their accounts receivable (money owed to them) from becoming too large. In response to a prolonged recession, one company has cut its customers’ credit limits in half. Thus, if a particular customer had a credit limit of $2000, it’s now $1000. If a customer had a credit limit of $5000, it’s now $2500. Write a program that analyzes the credit status of three customers of this company. For each customer you’re given:

a) The customer’s account number
b) The customer’s credit limit before the recession
c) The customer’s current balance (i.e., the amount the customer owes the company).

Your program should calculate and print the new credit limit for each customer and should determine (and print) which customers have current balances that exceed their new credit limits.

4.18  *(Bar Chart Printing Program)* One interesting application of computers is drawing graphs and bar charts (sometimes called “histograms”). Write a program that reads five numbers (each between 1 and 30). For each number read, your program should print a line containing that number of adjacent asterisks. For example, if your program reads the number seven, it should print *******.

4.19  *(Calculating Sales)* An online retailer sells five different products whose retail prices are shown in the following table:
Write a program that reads a series of pairs of numbers as follows:
   a) Product number
   b) Quantity sold for one day

Your program should use a switch statement to help determine the retail price for each product. Your program should calculate and display the total retail value of all products sold last week.

### 4.20 (Truth Tables)

Complete the following truth tables by filling in each blank with 0 or 1.

#### Condition 1 | Condition 2 | Condition 1 && Condition 2
---|---|---
0 | 0 | 0
0 | nonzero | 0
nonzero | 0 | ____
nonzero | nonzero | ____

#### Condition 1 | Condition 2 | Condition 1 || Condition 2
---|---|---
0 | 0 | 0
0 | nonzero | 1
nonzero | 0 | ____
nonzero | nonzero | ____

#### Condition 1 | ! Condition 1
---|---
0 | 1
nonzero | ____

### 4.21
Rewrite the program of Fig. 4.2 so that the initialization of the variable counter is done in the definition rather than in the for statement.

### 4.22 (Average Grade)
Modify the program of Fig. 4.7 so that it calculates the average grade for the class.

### 4.23 (Calculating the Compound Interest with Integers)
Modify the program of Fig. 4.6 so that it uses only integers to calculate the compound interest. \[Hint: \] Treat all monetary amounts as inte-
Exercises

4.24 Assume \( i = 1, j = 2, k = 3 \) and \( m = 2 \). What does each of the following statements print?

a) printf( "%d", i == 1 );

b) printf( "%d", j == 3 );

c) printf( "%d", i >= 1 && j < 4 );

d) printf( "%d", m <= 99 && k < m );

e) printf( "%d", j >= i || k == m );

f) printf( "%d", k + m < j || 3 - j >= k );

g) printf( "%d", !m );

h) printf( "%d", !( j - m ) );

i) printf( "%d", !( k > m ) );

j) printf( "%d", !( j > k ) );

4.25 (Table of Decimal, Binary, Octal and Hexadecimal Equivalents) Write a program that prints a table of the binary, octal and hexadecimal equivalents of the decimal numbers in the range 1 through 256. If you’re not familiar with these number systems, read Appendix C before you attempt this exercise.

4.26 (Calculating the Value of \( \pi \)) Calculate the value of \( \pi \) from the infinite series

\[
\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots
\]

Print a table that shows the value of \( \pi \) approximated by one term of this series, by two terms, by three terms, and so on. How many terms of this series do you have to use before you first get 3.14? 3.141? 3.1415? 3.14159?

4.27 (Pythagorean Triples) A right triangle can have sides that are all integers. The set of three integer values for the sides of a right triangle is called a Pythagorean triple. These three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Find all Pythagorean triples for side1, side2, and the hypotenuse all no larger than 500. Use a triple-nested for loop that simply tries all possibilities. This is an example of “brute-force” computing. It’s not aesthetically pleasing to many people. But there are many reasons why these techniques are important. First, with computing power increasing at such a phenomenal pace, solutions that would have taken years or even centuries of computer time to produce with the technology of just a few years ago can now be produced in hours, minutes or even seconds. Recent microprocessor chips can process a billion instructions per second! Second, as you’ll learn in more advanced computer science courses, there are large numbers of interesting problems for which there is no known algorithmic approach other than sheer brute force. We investigate many kinds of problem-solving methodologies in this book. We’ll consider many brute-force approaches to various interesting problems.

4.28 (Calculating Weekly Pay) A company pays its employees as managers (who receive a fixed weekly salary), hourly workers (who receive a fixed hourly wage for up to the first 40 hours they work and “time-and-a-half”—i.e., 1.5 times their hourly wage—for overtime hours worked), commission workers (who receive $250 plus 5.7% of their gross weekly sales), or pieceworkers (who receive a fixed amount of money for each of the items they produce—each pieceworker in this company works on only one type of item). Write a program to compute the weekly pay for each employee. You do not know the number of employees in advance. Each type of employee has its own pay code: Managers have paycode 1, hourly workers have code 2, commission workers have code 3 and pieceworkers have code 4. Use a switch to compute each employee’s pay based on that employee’s paycode. Within the switch, prompt the user (i.e., the payroll clerk) to enter the appropriate facts your program needs to calculate each employee’s pay based on that employee’s paycode.
4.29 (De Morgan’s Laws) In this chapter, we discussed the logical operators &&, ||, and !. De Morgan’s Laws can sometimes make it more convenient for us to express a logical expression. These laws state that the expression !(condition1 && condition2) is logically equivalent to the expression (!condition1 || !condition2). Also, the expression !(condition1 || condition2) is logically equivalent to the expression (!condition1 && !condition2). Use De Morgan’s Laws to write equivalent expressions for each of the following, and then write a program to show that both the original expression and the new expression in each case are equivalent.

   a) !( x < 5 ) && !( y >= 7 )
   b) !( a == b ) || !( g != 5 )
   c) !( ( x <= 8 ) && ( y > 4 ) )
   d) !( ( i > 4 ) || ( j <= 6 ) )

4.30 (Replacing switch with if…else) Rewrite the program of Fig. 4.7 by replacing the switch statement with a nested if…else statement; be careful to deal with the default case properly. Then rewrite this new version by replacing the nested if…else statement with a series of if statements; here, too, be careful to deal with the default case properly (this is more difficult than in the nested if…else version). This exercise demonstrates that switch is a convenience and that any switch statement can be written with only single-selection statements.

4.31 (Diamond Printing Program) Write a program that prints the following diamond shape. You may use printf statements that print either a single asterisk (*) or a single blank. Maximize your use of repetition (with nested for statements) and minimize the number of printf statements.

```
  *
 ***
 *****
*******
*********
 ********
  ****
  ***
   *
```

4.32 (Modified Diamond Printing Program) Modify the program you wrote in Exercise 4.31 to read an odd number in the range 1 to 19 to specify the number of rows in the diamond. Your program should then display a diamond of the appropriate size.

4.33 (Roman Numeral Equivalent of Decimal Values) Write a program that prints a table of all the Roman numeral equivalents of the decimal numbers in the range 1 to 100.

4.34 Describe the process you would use to replace a do…while loop with an equivalent while loop. What problem occurs when you try to replace a while loop with an equivalent do…while loop? Suppose you have been told that you must remove a while loop and replace it with a do…while. What additional control statement would you need to use and how would you use it to ensure that the resulting program behaves exactly as the original?

4.35 A criticism of the break statement and the continue statement is that each is unstructured. Actually, break statements and continue statements can always be replaced by structured statements, although doing so can be awkward. Describe in general how you would remove any break statement from a loop in a program and replace that statement with some structured equivalent. [Hint: The break statement leaves a loop from within the body of the loop. The other way to leave is by failing the loop-continuation test. Consider using in the loop-continuation test a second test that indicates “early exit because of a ‘break’ condition.”] Use the technique you developed here to remove the break statement from the program of Fig. 4.11.
4.36 What does the following program segment do?

```c
for ( i = 1; i <= 5; i++ ) {
    for ( j = 1; j <= 3; j++ ) {
        for ( k = 1; k <= 4; k++ )
            printf("*");
        printf("\n");
    }
    printf("\n");
}
```

4.37 Describe in general how you would remove any `continue` statement from a loop in a program and replace that statement with some structured equivalent. Use the technique you developed here to remove the `continue` statement from the program of Fig. 4.12.

4.38 *(World Population Growth)* World population has grown considerably over the centuries. Continued growth could eventually challenge the limits of breathable air, drinkable water, arable cropland and other limited resources. There is evidence that growth has been slowing in recent years and that world population could peak some time this century, then start to decline.

For this exercise, research world population growth issues online. *Be sure to investigate various viewpoints.* Get estimates for the current world population and its growth rate (the percentage by which it's likely to increase this year). Write a program that calculates world population growth each year for the next 75 years, *using the simplifying assumption that the current growth rate will stay constant.* Print the results in a table. The first column should display the year from year 1 to year 75. The second column should display the anticipated world population at the end of that year. The third column should display the numerical increase in the world population that would occur that year. Using your results, determine the year in which the population would be double what it is today, if this year's growth rate were to persist.

4.39 *(Tax Plan Alternatives; The "FairTax")* There are many proposals to make taxation fairer. Check out the FairTax initiative in the United States at

`www.fairtax.org/site/PageServer?pagename=calculator`

Research how the proposed FairTax works. One suggestion is to eliminate income taxes and most other taxes in favor of a 23% consumption tax on all products and services that you buy. Some FairTax opponents question the 23% figure and say that because of the way the tax is calculated, it would be more accurate to say the rate is 30%—check this carefully. Write a program that prompts the user to enter expenses in various expense categories they have (e.g., housing, food, clothing, transportation, education, health care, vacations), then prints the estimated FairTax that person would pay.
C Functions

Form ever follows function.
—Louis Henri Sullivan

E pluribus unum. (One composed of many.)
—Virgil

O! call back yesterday, bid time return.
—William Shakespeare

Answer me in one word.
—William Shakespeare

There is a point at which methods devour themselves.
—Frantz Fanon

Objectives
In this chapter, you’ll learn:

■ To construct programs modularly from small pieces called functions.

■ Common math functions in the C Standard Library.

■ To create new functions.

■ The mechanisms used to pass information between functions.

■ How the function call/return mechanism is supported by the function call stack and activation records.

■ Simulation techniques using random number generation.

■ How to write and use functions that call themselves.
5.1 Introduction

Most computer programs that solve real-world problems are much larger than the programs presented in the first few chapters. Experience has shown that the best way to develop and maintain a large program is to construct it from smaller pieces or modules, each of which is more manageable than the original program. This technique is called divide and conquer. This chapter describes the features of the C language that facilitate the design, implementation, operation and maintenance of large programs.

5.2 Program Modules in C

Modules in C are called functions. C programs are typically written by combining new functions you write with “prepackaged” functions available in the C Standard Library. We discuss both kinds of functions in this chapter. The C Standard Library provides a rich collection of functions for performing common mathematical calculations, string manipulations, character manipulations, input/output, and many other useful operations. This makes your job easier, because these functions provide many of the capabilities you need.

Good Programming Practice 5.1

Familiarize yourself with the rich collection of functions in the C Standard Library.

Software Engineering Observation 5.1

Avoid reinventing the wheel. When possible, use C Standard Library functions instead of writing new functions. This can reduce program development time.

Portability Tip 5.1

Using the functions in the C Standard Library helps make programs more portable.

Although the Standard Library functions are technically not a part of the C language, they’re provided with standard C systems. The functions printf, scanf and pow that we’ve used in previous chapters are Standard Library functions.
You can write functions to define specific tasks that may be used at many points in a program. These are sometimes referred to as **programmer-defined functions**. The actual statements defining the function are written only once, and the statements are hidden from other functions.

Functions are **invoked** by a **function call**, which specifies the function name and provides information (as **arguments**) that the called function needs to perform its designated task. A common analogy for this is the hierarchical form of management. A boss (the **calling function** or **caller**) asks a worker (the **called function**) to perform a task and report back when the task is done (Fig. 5.1). For example, a function needing to display information on the screen calls the worker function **printf** to perform that task, then **printf** displays the information and reports back—or **returns**—to the calling function when its task is completed. The boss function does not know how the worker function performs its designated tasks. The worker may call other worker functions, and the boss will be unaware of this. We’ll soon see how this “hiding” of implementation details promotes good software engineering. Figure 5.1 shows the **main** function communicating with several worker functions in a hierarchical manner. Note that worker1 acts as a boss function to worker4 and worker5. Relationships among functions may differ from the hierarchical structure shown in this figure.

![Fig. 5.1](image-url)  
**Fig. 5.1** | Hierarchical boss function/worker function relationship.

### 5.3 Math Library Functions

Math library functions allow you to perform certain common mathematical calculations. We use various math library functions here to introduce the concept of functions. Later in the book, we’ll discuss many of the other functions in the C Standard Library.

Functions are normally used in a program by writing the name of the function followed by a left parenthesis followed by the **argument** (or a comma-separated list of arguments) of the function followed by a right parenthesis. For example, a programmer desiring to calculate and print the square root of 900.0 might write

```c
printf( "%.2f", sqrt( 900.0 ) );
```

When this statement executes, the math library function **sqrt** is called to calculate the square root of the number contained in the parentheses (900.0). The number 900.0 is the
5.3 Math Library Functions

The argument of the sqrt function. The preceding statement would print 30.00. The sqrt function takes an argument of type double and returns a result of type double. All functions in the math library that return floating point values return the data type double. Note that double values, like float values, can be output using the %f conversion specification.

Function arguments may be constants, variables, or expressions. If \( c1 = 13.0 \), \( d = 3.0 \) and \( f = 4.0 \), then the statement

```c
printf( "%.2f", sqrt( c1 + d * f ) );
```

calculates and prints the square root of \( 13.0 + 3.0 \times 4.0 = 25.0 \), namely 5.00.

Some C math library functions are summarized in Fig. 5.2. In the figure, the variables \( x \) and \( y \) are of type double.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt( ( x ) )</td>
<td>square root of ( x )</td>
<td>sqrt( 900.0 ) is 30.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sqrt( 9.0 ) is 3.0</td>
</tr>
<tr>
<td>exp( ( x ) )</td>
<td>exponential function ( e^x )</td>
<td>exp( 1.0 ) is 2.718282</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exp( 2.0 ) is 7.389056</td>
</tr>
<tr>
<td>log( ( x ) )</td>
<td>natural logarithm of ( x ) (base ( e ))</td>
<td>log( 2.718282 ) is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log( 7.389056 ) is 2.0</td>
</tr>
<tr>
<td>log10( ( x ) )</td>
<td>logarithm of ( x ) (base 10)</td>
<td>log10( 1.0 ) is 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log10( 10.0 ) is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log10( 100.0 ) is 2.0</td>
</tr>
<tr>
<td>fabs( ( x ) )</td>
<td>absolute value of ( x )</td>
<td>fabs( 13.5 ) is 13.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fabs( 0.0 ) is 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fabs( -13.5 ) is 13.5</td>
</tr>
<tr>
<td>ceil( ( x ) )</td>
<td>rounds ( x ) to the smallest integer not less than ( x )</td>
<td>ceil( 9.2 ) is 10.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ceil( -9.8 ) is -9.0</td>
</tr>
<tr>
<td>floor( ( x ) )</td>
<td>rounds ( x ) to the largest integer not greater than ( x )</td>
<td>floor( 9.2 ) is 9.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>floor( -9.8 ) is -10.0</td>
</tr>
<tr>
<td>pow( ( x, y ) )</td>
<td>( x ) raised to power ( y ) ( (x^y) )</td>
<td>pow( 2, 7 ) is 128.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pow( 9, .5 ) is 3.0</td>
</tr>
<tr>
<td>fmod( ( x, y ) )</td>
<td>remainder of ( x/y ) as a floating-point number</td>
<td>fmod( 13.657, 2.333 ) is 1.992</td>
</tr>
<tr>
<td>sin( ( x ) )</td>
<td>trigonometric sine of ( x ) ( (x ) in radians)</td>
<td>sin( 0.0 ) is 0.0</td>
</tr>
<tr>
<td>cos( ( x ) )</td>
<td>trigonometric cosine of ( x ) ( (x ) in radians)</td>
<td>cos( 0.0 ) is 1.0</td>
</tr>
<tr>
<td>tan( ( x ) )</td>
<td>trigonometric tangent of ( x ) ( (x ) in radians)</td>
<td>tan( 0.0 ) is 0.0</td>
</tr>
</tbody>
</table>

Fig. 5.2 | Commonly used math library functions.
5.4 Functions

Functions allow you to modularize a program. All variables defined in function definitions are local variables—they’re known only in the function in which they’re defined. Most functions have a list of parameters that provide the means for communicating information between functions. A function’s parameters are also local variables of that function.

There are several motivations for “functionalizing” a program. The divide-and-conquer approach makes program development more manageable. Another motivation is software reusability—using existing functions as building-blocks to create new programs. Software reusability is a major factor in the object-oriented programming movement that you’ll learn more about when you study languages derived from C, such as C++, Java and C# (pronounced “C sharp”). With good function naming and definition, programs can be created from standardized functions that accomplish specific tasks, rather than being built by using customized code. This is known as abstraction. We use abstraction each time we use standard library functions like printf, scanf and pow. A third motivation is to avoid repeating code in a program. Packaging code as a function allows the code to be executed from several locations in a program simply by calling the function.

5.5 Function Definitions

Each program we’ve presented has consisted of a function called main that called standard library functions to accomplish its tasks. We now consider how to write custom functions. Consider a program that uses a function square to calculate and print the squares of the integers from 1 to 10 (Fig. 5.3).

Software Engineering Observation 5.2

In programs containing many functions, main is often implemented as a group of calls to functions that perform the bulk of the program’s work.

There are several motivations for “functionalizing” a program. The divide-and-conquer approach makes program development more manageable. Another motivation is software reusability—using existing functions as building-blocks to create new programs. Software reusability is a major factor in the object-oriented programming movement that you’ll learn more about when you study languages derived from C, such as C++, Java and C# (pronounced “C sharp”). With good function naming and definition, programs can be created from standardized functions that accomplish specific tasks, rather than being built by using customized code. This is known as abstraction. We use abstraction each time we use standard library functions like printf, scanf and pow. A third motivation is to avoid repeating code in a program. Packaging code as a function allows the code to be executed from several locations in a program simply by calling the function.

Software Engineering Observation 5.3

Each function should be limited to performing a single, well-defined task, and the function name should express that task. This facilitates abstraction and promotes software reusability.

Software Engineering Observation 5.4

If you cannot choose a concise name that expresses what the function does, it’s possible that your function is attempting to perform too many diverse tasks. It’s usually best to break such a function into several smaller functions—sometimes called decomposition.

Good Programming Practice 5.2

Place a blank line between function definitions to separate the functions and enhance program readability.

/* Fig. 5.3: fig05_03.c
   Creating and using a programmer-defined function */
#include <stdio.h>

Fig. 5.3 | Using a programmer-defined function. (Part 1 of 2.)
5.5 Function Definitions

Function square is **invoked** or **called** in main within the printf statement (line 14)

```c
printf("%d ", square(x)); /* function call */
```

Function square receives a copy of the value of x in the parameter y (line 22). Then square calculates $y \times y$ (line 24). The result is passed back to function printf in main where square was invoked (line 14), and printf displays the result. This process is repeated 10 times using the for repetition statement.

The definition of function square shows that square expects an integer parameter $y$. The keyword `int` preceding the function name (line 22) indicates that square returns an integer result. The return statement in square passes the result of the calculation back to the calling function.

Line 5

```c
int square(int y); /* function prototype */
```

is a function prototype. The `int` in parentheses informs the compiler that square expects to receive an integer value from the caller. The `int` to the left of the function name `square` informs the compiler that square returns an integer result to the caller. The compiler refers to the function prototype to check that calls to `square` (line 14) contain the correct return type, the correct number of arguments, the correct argument types, and that the arguments are in the correct order. Function prototypes are discussed in detail in Section 5.6.
The format of a function definition is

```
return-value-type function-name( parameter-list )
{
  definitions
  statements
}
```

The `function-name` is any valid identifier. The `return-value-type` is the data type of the result returned to the caller. The `return-value-type` `void` indicates that a function does not return a value. Together, the `return-value-type, function-name` and `parameter-list` are sometimes referred to as the function header.

The `parameter-list` is a comma-separated list that specifies the parameters received by the function when it’s called. If a function does not receive any values, `parameter-list` is `void`. A type must be listed explicitly for each parameter.

```
return-value-type function-name( parameter-list )
{
  definitions
  statements
}
```

The `parameter-list` is a comma-separated list that specifies the parameters received by the function when it’s called. If a function does not receive any values, `parameter-list` is `void`. A type must be listed explicitly for each parameter.

```
return-value-type function-name( parameter-list )
{
  definitions
  statements
}
```

Common Programming Error 5.1

*Forgetting to return a value from a function that is supposed to return a value can lead to unexpected errors. The C standard states that the result of this omission is undefined.*

Common Programming Error 5.2

*Returning a value from a function with a void return type is a compilation error.*

Common Programming Error 5.3

*Specifying function parameters of the same type as double x, y instead of double x, double y results in a compilation error.*

Common Programming Error 5.4

*Placing a semicolon after the right parenthesis enclosing the parameter list of a function definition is a syntax error.*

Common Programming Error 5.5

*Defining a parameter again as a local variable in a function is a compilation error.*

Good Programming Practice 5.3

*Although it’s not incorrect to do so, do not use the same names for a function’s arguments and the corresponding parameters in the function definition. This helps avoid ambiguity.*

The definitions and statements within braces form the function body. The function body is also referred to as a block. Variables can be declared in any block, and blocks can be nested. A function cannot be defined inside another function.

Good Programming Practice 5.4

*Choosing meaningful function names and meaningful parameter names makes programs more readable and helps avoid excessive use of comments.*
There are three ways to return control from a called function to the point at which a function was invoked. If the function does not return a result, control is returned simply when the function-ending right brace is reached, or by executing the statement

```
return;
```

If the function does return a result, the statement

```
return expression;
```

returns the value of expression to the caller.

**Function maximum**

Our second example uses a programmer-defined function maximum to determine and return the largest of three integers (Fig. 5.4). The three integers are input with scanf\(^1\) (line 15). Next, the integers are passed to maximum (line 19), which determines the largest integer. This value is returned to main by the return statement in maximum (line 37). The value returned is then printed in the printf statement (line 19).

---

1. Many C library functions, like scanf, return values indicating whether they performed their task successfully. In production code, you should test these return values to ensure that your program is operating properly. Read the documentation for each library function you use to learn about its return values. The site wpollock.com/CPlus/PrintfRef.htm#scanfRetCode discusses how to process return values from function scanf.
Chapter 5  C Functions

5.6 Function Prototypes

One of the most important features of C is the function prototype. This feature was bor-rowed by the C standard committee from the developers of C++. A function prototype tells the compiler the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters, and the order in which these parameters are expected. The compiler uses function prototypes to validate function calls.
Previous versions of C did not perform this kind of checking, so it was possible to call functions improperly without the compiler detecting the errors. Such calls could result in fatal execution-time errors or nonfatal errors that caused subtle, difficult-to-detect logic errors. Function prototypes correct this deficiency.

### Good Programming Practice 5.5
Include function prototypes for all functions to take advantage of C’s type-checking capabilities. Use `#include` preprocessor directives to obtain function prototypes for the standard library functions from the headers for the appropriate libraries, or to obtain headers containing function prototypes for functions developed by you and/or your group members.

The function prototype for `maximum` in Fig. 5.4 (line 5) is

```c
int maximum(int x, int y, int z); /* function prototype */
```

This function prototype states that `maximum` takes three arguments of type `int` and returns a result of type `int`. Notice that the function prototype is the same as the first line of the function definition of `maximum`.

### Good Programming Practice 5.6
Parameter names are sometimes included in function prototypes (our preference) for documentation purposes. The compiler ignores these names.

### Common Programming Error 5.7
Forgetting the semicolon at the end of a function prototype is a syntax error.

A function call that does not match the function prototype is a compilation error. An error is also generated if the function prototype and the function definition disagree. For example, in Fig. 5.4, if the function prototype had been written

```c
void maximum(int x, int y, int z);
```

the compiler would generate an error because the `void` return type in the function prototype would differ from the `int` return type in the function header.

Another important feature of function prototypes is the **coercion of arguments**, i.e., the forcing of arguments to the appropriate type. For example, the math library function `sqrt` can be called with an integer argument even though the function prototype in `<math.h>` specifies a `double` argument, and the function will still work correctly. The statement

```c
printf("%.3f\n", sqrt(4));
```

correctly evaluates `sqrt(4)`, and prints the value 2.000. The function prototype causes the compiler to convert the integer value 4 to the `double` value 4.0 before the value is passed to `sqrt`. In general, argument values that do not correspond precisely to the parameter types in the function prototype are converted to the proper type before the function is called. These conversions can lead to incorrect results if C’s **promotion rules** are not followed. The promotion rules specify how types can be converted to other types without losing data. In our `sqrt` example above, an `int` is automatically converted to a `double`
without changing its value. However, a `double` converted to an `int` truncates the fractional part of the `double` value. Converting large integer types to small integer types (e.g., `long` to `short`) may also result in changed values.

The promotion rules automatically apply to expressions containing values of two or more data types (also referred to as **mixed-type expressions**). The type of each value in a mixed-type expression is automatically promoted to the “highest” type in the expression (actually a temporary version of each value is created and used for the expression—the original values remain unchanged). Figure 5.5 lists the data types in order from highest type to lowest type with each type’s `printf` and `scanf` conversion specifications.

<table>
<thead>
<tr>
<th>Data type</th>
<th><code>printf</code> conversion specification</th>
<th><code>scanf</code> conversion specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>long double</code></td>
<td><code>%Lf</code></td>
<td><code>%Lf</code></td>
</tr>
<tr>
<td><code>double</code></td>
<td><code>%f</code></td>
<td><code>%lf</code></td>
</tr>
<tr>
<td><code>float</code></td>
<td><code>%f</code></td>
<td><code>%f</code></td>
</tr>
<tr>
<td><code>unsigned long int</code></td>
<td><code>%lu</code></td>
<td><code>%lu</code></td>
</tr>
<tr>
<td><code>long int</code></td>
<td><code>%ld</code></td>
<td><code>%ld</code></td>
</tr>
<tr>
<td><code>unsigned int</code></td>
<td><code>%u</code></td>
<td><code>%u</code></td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>%d</code></td>
<td><code>%d</code></td>
</tr>
<tr>
<td><code>unsigned short</code></td>
<td><code>%hu</code></td>
<td><code>%hu</code></td>
</tr>
<tr>
<td><code>short</code></td>
<td><code>%hd</code></td>
<td><code>%hd</code></td>
</tr>
<tr>
<td><code>char</code></td>
<td><code>%c</code></td>
<td><code>%c</code></td>
</tr>
</tbody>
</table>

**Fig. 5.5** | Promotion hierarchy for data types.

Converting values to lower types normally results in an incorrect value. Therefore, a value can be converted to a lower type only by explicitly assigning the value to a variable of lower type, or by using a cast operator. Function argument values are converted to the parameter types in a function prototype as if they were being assigned directly to variables of those types. If our `square` function that uses an integer parameter (Fig. 5.3) is called with a floating-point argument, the argument is converted to `int` (a lower type), and `square` usually returns an incorrect value. For example, `square( 4.5 )` returns 16, not 20.25.

**Common Programming Error 5.8**

*Converting from a higher data type in the promotion hierarchy to a lower type can change the data value. Many compilers issue warnings in such cases.*

If there is no function prototype for a function, the compiler forms its own function prototype using the first occurrence of the function—either the function definition or a call to the function. This typically leads to warnings or errors, depending on the compiler.

**Error-Prevention Tip 5.2**

*Always include function prototypes for the functions you define or use in your program to help prevent compilation errors and warnings.*
5.7 Function Call Stack and Activation Records

To understand how C performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a **stack**. Students can think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it’s normally placed at the top (referred to as **pushing** the dish onto the stack). Similarly, when a dish is removed from the pile, it’s always removed from the top (referred to as **popping** the dish off the stack). Stacks are known as **last-in, first-out (LIFO)** data structures—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

When a program calls a function, the called function must know how to return to its caller, so the return address of the calling function is pushed onto the **program execution stack** (sometimes referred to as the **function call stack**). If a series of function calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order so that each function can return to its caller.

The program execution stack also contains the memory for the local variables used in each invocation of a function during a program’s execution. This data, stored as a portion of the program execution stack, is known as the **activation record** or **stack frame** of the function call. When a function call is made, the activation record for that function call is pushed onto the program execution stack. When the function returns to its caller, the activation record for this function call is popped off the stack and those local variables are no longer known to the program.

Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the program execution stack. If more function calls occur than can have their activation records stored on the program execution stack, an error known as a **stack overflow** occurs.

5.8 Headers

Each standard library has a corresponding **header** containing the function prototypes for all the functions in that library and definitions of various data types and constants needed by those functions. Figure 5.6 lists alphabetically some of the standard library headers that may be included in programs. The term “macros” that is used several times in Fig. 5.6 is discussed in detail in Chapter 13, C Preprocessor.

You can create custom headers. Programmer-defined headers should also use the .h filename extension. A programmer-defined header can be included by using the **#include** preprocessor directive. For example, if the prototype for our square function was located in the header square.h, we’d include that header in our program by using the following directive at the top of the program:

```
#include "square.h"
```

Section 13.2 presents additional information on including headers.
5.9 Calling Functions By Value and By Reference

There are two ways to invoke functions in many programming languages—call-by-value and call-by-reference. When arguments are passed by value, a copy of the argument’s value is made and passed to the called function. Changes to the copy do not affect an original variable’s value in the caller. When an argument is passed by reference, the caller allows the called function to modify the original variable.

Call-by-value should be used whenever the called function does not need to modify the value of the caller’s original variable. This prevents the accidental side effects (variable modifications) that so greatly hinder the development of correct and reliable software systems. Call-by-reference should be used only with trusted called functions that need to modify the original variable.

In C, all calls are by value. As we’ll see in Chapter 7, it’s possible to simulate call-by-reference by using address operators and indirection operators. In Chapter 6, we’ll see that

<table>
<thead>
<tr>
<th>Header</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;assert.h&gt;</td>
<td>Contains macros and information for adding diagnostics that aid program debugging.</td>
</tr>
<tr>
<td>&lt;ctype.h&gt;</td>
<td>Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.</td>
</tr>
<tr>
<td>&lt;errno.h&gt;</td>
<td>Defines macros that are useful for reporting error conditions.</td>
</tr>
<tr>
<td>&lt;float.h&gt;</td>
<td>Contains the floating-point size limits of the system.</td>
</tr>
<tr>
<td>&lt;limits.h&gt;</td>
<td>Contains the integral size limits of the system.</td>
</tr>
<tr>
<td>&lt;locale.h&gt;</td>
<td>Contains function prototypes and other information that enables a program to be modified for the current locale on which it’s running. The notion of locale enables the computer system to handle different conventions for expressing data like dates, times, dollar amounts and large numbers throughout the world.</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>Contains function prototypes for math library functions.</td>
</tr>
<tr>
<td>&lt;setjmp.h&gt;</td>
<td>Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.</td>
</tr>
<tr>
<td>&lt;signal.h&gt;</td>
<td>Contains function prototypes and macros to handle various conditions that may arise during program execution.</td>
</tr>
<tr>
<td>&lt;stdarg.h&gt;</td>
<td>Defines macros for dealing with a list of arguments to a function whose number and types are unknown.</td>
</tr>
<tr>
<td>&lt;stddef.h&gt;</td>
<td>Contains common type definitions used by C for performing calculations.</td>
</tr>
<tr>
<td>&lt;stdio.h&gt;</td>
<td>Contains function prototypes for the standard input/output library functions, and information used by them.</td>
</tr>
<tr>
<td>&lt;stdlib.h&gt;</td>
<td>Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.</td>
</tr>
<tr>
<td>&lt;string.h&gt;</td>
<td>Contains function prototypes for string-processing functions.</td>
</tr>
<tr>
<td>&lt;time.h&gt;</td>
<td>Contains function prototypes and types for manipulating the time and date.</td>
</tr>
</tbody>
</table>
arrays are automatically passed by reference. We’ll have to wait until Chapter 7 for a full understanding of this complex issue. For now, we concentrate on call-by-value.

5.10 Random Number Generation

We now take a brief and, hopefully, entertaining diversion into a popular programming application, namely simulation and game playing. In this and the next section, we’ll develop a nicely structured game-playing program that includes multiple functions. The program uses most of the control structures we’ve studied. The element of chance can be introduced into computer applications by using the C Standard Library function rand from the <stdlib.h> header.

Consider the following statement:

The rand function generates an integer between 0 and RAND_MAX (a symbolic constant defined in the <stdlib.h> header). Standard C states that the value of RAND_MAX must be at least 32767, which is the maximum value for a two-byte (i.e., 16-bit) integer. The programs in this section were tested on a C system with a maximum value of 32767 for RAND_MAX. If rand truly produces integers at random, every number between 0 and RAND_MAX has an equal chance (or probability) of being chosen each time rand is called.

The range of values produced directly by rand is often different from what is needed in a specific application. For example, a program that simulates coin tossing might require only 0 for “heads” and 1 for “tails.” A dice-rolling program that simulates a six-sided die would require random integers from 1 to 6.

Rolling a Six-Sided Die

To demonstrate rand, let’s develop a program to simulate 20 rolls of a six-sided die and print the value of each roll. The function prototype for function rand is in <stdlib.h>.

We use the remainder operator (\%) in conjunction with rand as follows to produce integers in the range 0 to 5. This is called scaling. The number 6 is called the scaling factor. We then shift the range of numbers produced by adding 1 to our previous result. The output of Fig. 5.7 confirms that the results are in the range 1 to 6—the output might vary by compiler.

```c
/* Fig. 5.7: fig05_07.c */
Shifted, scaled integers produced by 1 + rand() % 6 */
#include <stdio.h>
#include <stdlib.h>

/* function main begins program execution */
int main( void )
{
    int i; /* counter */
```

Fig. 5.7 | Shifted, scaled random integers produced by 1 + rand() % 6. (Part 1 of 2.)
Rolling a Six-Sided Die 6000 Times
To show that these numbers occur approximately with equal likelihood, let’s simulate 6000 rolls of a die with the program of Fig. 5.8. Each integer from 1 to 6 should appear approximately 1000 times.

As the program output shows, by scaling and shifting we’ve used the rand function to realistically simulate the rolling of a six-sided die. No default case is provided in the switch statement. Also note the use of the %s conversion specifier to print the character strings "Face" and "Frequency" as column headers (line 53). After we study arrays in Chapter 6, we’ll show how to replace this entire switch statement elegantly with a single-line statement.

```c
/* loop 20 times */
for ( i = 1; i <= 20; i++ ) {
    /* pick random number from 1 to 6 and output it */
    printf( "%10d", 1 + ( rand() % 6 ) );
    /* if counter is divisible by 5, begin new line of output */
    if ( i % 5 == 0 ) {
        printf( "\n" );
    } /* end if */
} /* end for */
return 0; /* indicates successful termination */
} /* end main */
```

Fig. 5.7 | Shifted, scaled random integers produced by $1 + \text{rand()} \% 6$. (Part 2 of 2.)

Fig. 5.8 | Rolling a six-sided die 6000 times. (Part 1 of 2.)
int face; /* represents one roll of the die, value 1 to 6 */

/* loop 6000 times and summarize results */
for ( roll = 1; roll <= 6000; roll++ ) {
  face = 1 + rand() % 6; /* random number from 1 to 6 */

  /* determine face value and increment appropriate counter */
  switch ( face ) {
    case 1: /* rolled 1 */
      ++frequency1;
      break;
    case 2: /* rolled 2 */
      ++frequency2;
      break;
    case 3: /* rolled 3 */
      ++frequency3;
      break;
    case 4: /* rolled 4 */
      ++frequency4;
      break;
    case 5: /* rolled 5 */
      ++frequency5;
      break;
    case 6: /* rolled 6 */
      ++frequency6;
      break; /* optional */
  } /* end switch */
} /* end for */

/* display results in tabular format */
printf( "%s%13s\n", "Face", "Frequency" );
printf( " %13d\n", frequency1 );
printf( " %13d\n", frequency2 );
printf( " %13d\n", frequency3 );
printf( " %13d\n", frequency4 );
printf( " %13d\n", frequency5 );
printf( " %13d\n", frequency6 );

return 0; /* indicates successful termination */
} /* end main */

<table>
<thead>
<tr>
<th>Face</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1003</td>
</tr>
<tr>
<td>2</td>
<td>1017</td>
</tr>
<tr>
<td>3</td>
<td>983</td>
</tr>
<tr>
<td>4</td>
<td>994</td>
</tr>
<tr>
<td>5</td>
<td>1004</td>
</tr>
<tr>
<td>6</td>
<td>999</td>
</tr>
</tbody>
</table>

Fig. 5.8 | Rolling a six-sided die 6000 times. (Part 2 of 2.)
Randomizing the Random Number Generator

Executing the program of Fig. 5.7 again produces

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Notice that exactly the same sequence of values was printed. How can these be random numbers? Ironically, this repeatability is an important characteristic of function `rand`. When debugging a program, this repeatability is essential for proving that corrections to a program work properly.

Function `rand` actually generates pseudorandom numbers. Calling `rand` repeatedly produces a sequence of numbers that appears to be random. However, the sequence repeats itself each time the program is executed. Once a program has been thoroughly debugged, it can be conditioned to produce a different sequence of random numbers for each execution. This is called randomizing and is accomplished with the standard library function `srand`. Function `srand` takes an unsigned integer argument and seeds function `rand` to produce a different sequence of random numbers for each execution of the program.

We demonstrate `srand` in Fig. 5.9. In the program, we use the data type `unsigned`, which is short for `unsigned int`. An `int` is stored in at least two bytes of memory and can have positive and negative values. A variable of type `unsigned` is also stored in at least two bytes of memory. A two-byte `unsigned int` can have only positive values in the range 0 to 65535. A four-byte `unsigned int` can have only positive values in the range 0 to 4294967295. Function `srand` takes an `unsigned` value as an argument. The conversion specifier `%u` is used to read an `unsigned` value with `scanf`. The function prototype for `srand` is found in `<stdlib.h>`.

---

```c
/* Fig. 5.9: fig05_09.c */
Randomizing die-rolling program */
#include <stdlib.h>
#include <stdio.h>
/* function main begins program execution */
int main( void )
{
    int i; /* counter */
    unsigned seed; /* number used to seed random number generator */
    printf( "Enter seed: " );
    scanf( "%u", &seed ); /* note %u for unsigned */
    srand( seed ); /* seed random number generator */
```

---

Fig. 5.9 | Randomizing the die-rolling program. (Part 1 of 2.)
Let’s run the program several times and observe the results. Notice that a different sequence of random numbers is obtained each time the program is run, provided that a different seed is supplied.

To randomize without entering a seed each time, use a statement like

```c
srand( time( NULL ) );
```

This causes the computer to read its clock to obtain the value for the seed automatically. Function `time` returns the number of seconds that have passed since midnight on January 1, 1970. This value is converted to an unsigned integer and used as the seed to the random number generator. Function `time` takes `NULL` as an argument (time is capable of providing you with a string representing the value it returns; `NULL` disables this capability for a specific call to `time`). The function prototype for `time` is in `<time.h>`.

**Generalized Scaling and Shifting of Random Numbers**

The values produced directly by `rand` are always in the range:

```c
0 \leq \text{rand()} \leq \text{RAND_MAX}
```
As you know, the following statement simulates rolling a six-sided die:

```c
face = 1 + rand() % 6;
```

This statement always assigns an integer value (at random) to the variable `face` in the range \(1 \leq \text{face} \leq 6\). The width of this range (i.e., the number of consecutive integers in the range) is 6 and the starting number in the range is 1. Referring to the preceding statement, we see that the width of the range is determined by the number used to scale `rand` with the remainder operator (i.e., 6), and the starting number of the range is equal to the number (i.e., 1) that is added to `rand % 6`. We can generalize this result as follows:

```c
n = a + rand() % b;
```

where `a` is the **shifting value** (which is equal to the first number in the desired range of consecutive integers) and `b` is the scaling factor (which is equal to the width of the desired range of consecutive integers). In the exercises, we’ll see that it’s possible to choose integers at random from sets of values other than ranges of consecutive integers.

### Common Programming Error 5.9

*Using `srand` in place of `rand` to generate random numbers.*

#### 5.11 Example: A Game of Chance

One of the most popular games of chance is a dice game known as “craps,” which is played in casinos and back alleys throughout the world. The rules of the game are straightforward:

A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5, and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3, or 12 on the first throw (called “craps”), the player loses (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, then that sum becomes the player’s “point.” To win, you must continue rolling the dice until you “make your point.” The player loses by rolling a 7 before making the point.

Figure 5.10 simulates the game of craps and Fig. 5.11 shows several sample executions.

```c
/* Fig. 5.10: fig05_10.c
Craps */
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* contains prototype for function time */
/* enumeration constants represent game status */
enum Status { CONTINUE, WON, LOST };
int rollDice( void ); /* function prototype */
/* function main begins program execution */
int main( void )
{

Fig. 5.10 | Program to simulate the game of craps. (Part 1 of 3.)
```
```c
int sum; /* sum of rolled dice */
int myPoint; /* point earned */

enum Status gameStatus; /* can contain CONTINUE, WON, or LOST */

/* randomize random number generator using current time */
srand( time( NULL ) );

sum = rollDice(); /* first roll of the dice */

/* determine game status based on sum of dice */
switch( sum ) {
    /* win on first roll */
    case 7:
    case 11:
        gameStatus = WON;
        break;
    /* lose on first roll */
    case 2:
    case 3:
    case 12:
        gameStatus = LOST;
        break;
    /* remember point */
    default:
        gameStatus = CONTINUE;
        myPoint = sum;
        printf( "Point is %d\n", myPoint );
        break; /* optional */
} /* end switch */

/* while game not complete */
while ( gameStatus == CONTINUE ) {
    sum = rollDice(); /* roll dice again */
    /* determine game status */
    if ( sum == myPoint ) { /* win by making point */
        gameStatus = WON; /* game over, player won */
    } /* end if */
    else {
        if ( sum == 7 ) { /* lose by rolling 7 */
            gameStatus = LOST; /* game over, player lost */
        } /* end if */
    } /* end else */
} /* end while */

/* display won or lost message */
if ( gameStatus == WON ) { /* did player win? */
    printf( "Player wins\n" );
} /* end if */
```

Fig. 5.10 | Program to simulate the game of craps. (Part 2 of 3.)
Chapter 5  C Functions

In the rules of the game, notice that the player must roll two dice on the first roll, and
must do so later on all subsequent rolls. We define a function \texttt{rollDice} to roll the dice
and compute and print their sum. Function \texttt{rollDice} is defined once, but it’s called from

```
68    \textbf{else} \ { /* player lost */
69     \texttt{printf( "Player loses\n" );}
70     \} /* end else */
71  
72     \texttt{return 0; /* indicates successful termination */}
73     \} /* end main */
74 
75  \} /* roll dice, calculate sum and display results */
76 \textbf{int} \ \texttt{rollDice( \textbf{void} )}
77  
78 \{ \texttt{int} \ \texttt{die1; /* first die */
79     \texttt{int} \ \texttt{die2; /* second die */
80     \texttt{int} \ \texttt{workSum; /* sum of dice */
81     \texttt{die1} = 1 + ( \texttt{rand()} \ % \ 6 ); /* pick random die1 value */
82     \texttt{die2} = 1 + ( \texttt{rand()} \ % \ 6 ); /* pick random die2 value */
83     \texttt{workSum} = \texttt{die1} + \texttt{die2}; /* sum die1 and die2 */
84     \} /* display results of this roll */
85     \texttt{printf( "Player rolled %d + %d = %d\n", \texttt{die1}, \texttt{die2}, \texttt{workSum} );}
86     \texttt{return workSum; /* return sum of dice */
87     \} /* end function rollDice */
```

\begin{verbatim}
Player rolled 5 + 6 = 11
Player wins

Player rolled 4 + 1 = 5
Point is 5
Player rolled 6 + 2 = 8
Player rolled 2 + 1 = 3
Player rolled 3 + 2 = 5
Player wins

Player rolled 1 + 1 = 2
Player loses

Player rolled 6 + 4 = 10
Point is 10
Player rolled 3 + 4 = 7
Player loses
\end{verbatim}

\textbf{Fig. 5.11} \hspace{1cm} Sample runs for the game of craps.

In the rules of the game, notice that the player must roll two dice on the first roll, and
must do so later on all subsequent rolls. We define a function \texttt{rollDice} to roll the dice
and compute and print their sum. Function \texttt{rollDice} is defined once, but it’s called from
two places in the program (lines 23 and 51). Interestingly, rollDice takes no arguments, so we’ve indicated void in the parameter list (line 76). Function rollDice does return the sum of the two dice, so a return type of int is indicated in the function header.

The game is reasonably involved. The player may win or lose on the first roll, or may win or lose on any subsequent roll. Variable gameStatus, defined to be of a new type—enum Status—stores the current status. Line 8 creates a programmer-defined type called an enumeration. An enumeration, introduced by the keyword enum, is a set of integer constants represented by identifiers. Enumeration constants are sometimes called symbolic constants. Values in an enum start with 0 and are incremented by 1. In line 8, the constant CONTINUE has the value 0, WON has the value 1 and LOST has the value 2. It’s also possible to assign an integer value to each identifier in an enum (see Chapter 10). The identifiers in an enumeration must be unique, but the values may be duplicated.

When the game is won, either on the first roll or on a subsequent roll, gameStatus is set to WON. When the game is lost, either on the first roll or on a subsequent roll, gameStatus is set to LOST. Otherwise gameStatus is set to CONTINUE and the game continues.

After the first roll, if the game is over, the while statement (line 50) is skipped because gameStatus is not CONTINUE. The program proceeds to the if…else statement at line 65, which prints "Player wins" if gameStatus is WON and "Player loses" otherwise.

After the first roll, if the game is not over, then sum is saved in myPoint. Execution proceeds with the while statement (line 50) because gameStatus is CONTINUE. Each time through the while, rollDice is called to produce a new sum. If sum matches myPoint, gameStatus is set to WON to indicate that the player won, the while-test fails, the if…else statement (line 65) prints "Player wins" and execution terminates. If sum is equal to 7 (line 58), gameStatus is set to LOST to indicate that the player lost, the while-test fails, the if…else statement (line 65) prints "Player loses" and execution terminates.

Note the program’s interesting control architecture. We’ve used two functions—main and rollDice—and the switch, while, nested if…else and nested if statements. In the exercises, we’ll investigate various interesting characteristics of the game of craps.

**5.12 Storage Classes**

In Chapters 2–4, we used identifiers for variable names. The attributes of variables include name, type, size and value. In this chapter, we also use identifiers as names for user-defined functions. Actually, each identifier in a program has other attributes, including storage class, storage duration, scope and linkage.

C provides four storage classes, indicated by the storage class specifiers: auto, register, extern and static. An identifier’s storage class determines its storage duration, scope and linkage. An identifier’s storage duration is the period during which the identifier exists in memory. Some exist briefly, some are repeatedly created and destroyed, and
others exist for the entire execution of a program. An identifier’s scope is where the identifier can be referenced in a program. Some can be referenced throughout a program, others from only portions of a program. An identifier’s linkage determines for a multiple-source-file program (a topic we’ll investigate in Chapter 14) whether the identifier is known only in the current source file or in any source file with proper declarations. This section discusses storage classes and storage duration. Section 5.13 discusses scope. Chapter 14 discusses identifier linkage and programming with multiple source files.

The four storage-class specifiers can be split into two storage durations: automatic storage duration and static storage duration. Keywords auto and register are used to declare variables of automatic storage duration. Variables with automatic storage duration are created when the block in which they’re defined is entered; they exist while the block is active, and they’re destroyed when the block is exited.

Local Variables
Only variables can have automatic storage duration. A function’s local variables (those declared in the parameter list or function body) normally have automatic storage duration. Keyword auto explicitly declares variables of automatic storage duration. For example, the following declaration indicates that double variables x and y are automatic local variables and they exist only in the body of the function in which the declaration appears:

```c
auto double x, y;
```

Local variables have automatic storage duration by default, so keyword auto is rarely used. For the remainder of the text, we’ll refer to variables with automatic storage duration simply as automatic variables.

**Performance Tip 5.1**
Automatic storage is a means of conserving memory, because automatic variables exist only when they’re needed. They’re created when a function is entered and destroyed when the function is exited.

**Software Engineering Observation 5.10**
Automatic storage is an example of the principle of least privilege—allowing access to data only when it’s absolutely needed. Why have variables stored in memory and accessible when in fact they’re not needed?

Register Variables
Data in the machine-language version of a program is normally loaded into registers for calculations and other processing.

**Performance Tip 5.2**
The storage-class specifier register can be placed before an automatic variable declaration to suggest that the compiler maintain the variable in one of the computer’s high-speed hardware registers. If intensely used variables such as counters or totals can be maintained in hardware registers, the overhead of repeatedly loading the variables from memory into the registers and storing the results back into memory can be eliminated.

The compiler may ignore register declarations. For example, there may not be a sufficient number of registers available for the compiler to use. The following declaration sug-
gests that the integer variable \texttt{counter} be placed in one of the computer’s registers and initialized to 1:

\begin{verbatim}
register int counter = 1;
\end{verbatim}

Keyword \texttt{register} can be used only with variables of automatic storage duration.

\begin{quote}
\textbf{Performance Tip 5.3}

Often, \textit{register} declarations are unnecessary. Today’s optimizing compilers are capable of recognizing frequently used variables and can decide to place them in registers without the need for a \textit{register} declaration.
\end{quote}

\textbf{Static Storage Class}

Keywords \texttt{extern} and \texttt{static} are used in the declarations of identifiers for variables and functions of static storage duration. Identifiers of static storage duration exist from the time at which the program begins execution. For static variables, storage is allocated and initialized once, when the program begins execution. For functions, the name of the function exists when the program begins execution. However, even though the variables and the function names exist from the start of program execution, this does not mean that these identifiers can be accessed throughout the program. Storage duration and scope (where a name can be used) are separate issues, as we’ll see in Section 5.13.

There are two types of identifiers with static storage duration: external identifiers (such as global variables and function names) and local variables declared with the storage-class specifier \texttt{static}. Global variables and function names are of storage class \texttt{extern} by default. Global variables are created by placing variable declarations outside any function definition, and they retain their values throughout the execution of the program. Global variables and functions can be referenced by any function that follows their declarations or definitions in the file. This is one reason for using function prototypes—when we include \texttt{stdio.h} in a program that calls \texttt{printf}, the function prototype is placed at the start of our file to make the name \texttt{printf} known to the rest of the file.

\begin{quote}
\textbf{Software Engineering Observation 5.11}

Defining a variable as global rather than local allows unintended side effects to occur when a function that does not need access to the variable accidentally or maliciously modifies it. In general, use of global variables should be avoided except in certain situations with unique performance requirements (as discussed in Chapter 14).
\end{quote}

\begin{quote}
\textbf{Software Engineering Observation 5.12}

Variables used only in a particular function should be defined as local variables in that function rather than as external variables.
\end{quote}

Local variables declared with the keyword \texttt{static} are still known only in the function in which they’re defined, but unlike automatic variables, \texttt{static} local variables retain their value when the function is exited. The next time the function is called, the \texttt{static} local variable contains the value it had when the function last exited. The following statement declares local variable \texttt{count} to be \texttt{static} and to be initialized to 1.

\begin{verbatim}
static int count = 1;
\end{verbatim}
All numeric variables of static storage duration are initialized to zero if you do not explicitly initialize them.

Keywords extern and static have special meaning when explicitly applied to external identifiers. In Chapter 14 we discuss the explicit use of extern and static with external identifiers and multiple-source-file programs.

5.13 Scope Rules

The scope of an identifier is the portion of the program in which the identifier can be referenced. For example, when we define a local variable in a block, it can be referenced only following its definition in that block or in blocks nested within that block. The four identifier scopes are function scope, file scope, block scope, and function-prototype scope.

Labels (an identifier followed by a colon such as start:) are the only identifiers with function scope. Labels can be used anywhere in the function in which they appear, but cannot be referenced outside the function body. Labels are used in switch statements (as case labels) and in goto statements (see Chapter 14). Labels are implementation details that functions hide from one another. This hiding—more formally called information hiding—is a means of implementing the principle of least privilege, one of the most fundamental principles of good software engineering.

An identifier declared outside any function has file scope. Such an identifier is “known” (i.e., accessible) in all functions from the point at which the identifier is declared until the end of the file. Global variables, function definitions, and function prototypes placed outside a function all have file scope.

Identifiers defined inside a block have block scope. Block scope ends at the terminating right brace (}) of the block. Local variables defined at the beginning of a function have block scope as do function parameters, which are considered local variables by the function. Any block may contain variable definitions. When blocks are nested, and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is “hidden” until the inner block terminates. This means that while executing in the inner block, the inner block sees the value of its own local identifier and not the value of the identically named identifier in the enclosing block. Local variables declared static still have block scope, even though they exist from the time the program begins execution. Thus, storage duration does not affect the scope of an identifier.

The only identifiers with function-prototype scope are those used in the parameter list of a function prototype. As mentioned previously, function prototypes do not require names in the parameter list—only types are required. If a name is used in the parameter list of a function prototype, the compiler ignores the name. Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity.

Common Programming Error 5.12

Accidentally using the same name for an identifier in an inner block as is used for an identifier in an outer block, when in fact you want the identifier in the outer block to be active for the duration of the inner block.

Error-Prevention Tip 5.3

Avoid variable names that hide names in outer scopes. This can be accomplished simply by avoiding the use of duplicate identifiers in a program.
Figure 5.12 demonstrates scoping issues with global variables, automatic local variables, and static local variables. A global variable $x$ is defined and initialized to 1 (line 9). This global variable is hidden in any block (or function) in which a variable named $x$ is defined. In main, a local variable $x$ is defined and initialized to 5 (line 14). This variable is then printed to show that the global $x$ is hidden in main. Next, a new block is defined in main with another local variable $x$ initialized to 7 (line 19). This variable is printed to show that it hides $x$ in the outer block of main. The variable $x$ with value 7 is automatically destroyed when the block is exited, and the local variable $x$ in the outer block of main is printed again to show that it’s no longer hidden. The program defines three functions that each take no arguments and return nothing. Function useLocal defines an automatic variable $x$ and initializes it to 25 (line 40). When useLocal is called, the variable is printed, incremented, and printed again before exiting the function. Each time this function is called, automatic variable $x$ is reinitialized to 25. Function useStaticLocal defines a static variable $x$ and initializes it to 50 (line 53). Local variables declared as static retain their values even when they’re out of scope. When useStaticLocal is called, $x$ is printed, incremented, and printed again before exiting the function. In the next call to this function, static local variable $x$ will contain the value 51. Function useGlobal does not define any variables. Therefore, when it refers to variable $x$, the global $x$ (line 9) is used. When useGlobal is called, the global variable is printed, multiplied by 10, and printed again before exiting the function. The next time function useGlobal is called, the global variable still has its modified value, 10. Finally, the program prints the local variable $x$ in main again (line 33) to show that none of the function calls modified the value of $x$ because the functions all referred to variables in other scopes.

```c
/* Fig. 5.12: fig05_12.c */
#include <stdio.h>

void useLocal( void ); /* function prototype */
void useStaticLocal( void ); /* function prototype */
void useGlobal( void ); /* function prototype */

int x = 1; /* global variable */

/* function main begins program execution */
main( void )
{
    int x = 5; /* local variable to main */
    printf("local x in outer scope of main is %d\n", x);
    {
        int x = 7; /* local variable to new scope */
        printf("local x in inner scope of main is %d\n", x);
    } /* end new scope */
    printf("local x in outer scope of main is %d\n", x);
}
```

**Fig. 5.12** Scoping example. (Part I of 3.)
function useLocal() /* useLocal has automatic local x */
function useStaticLocal() /* useStaticLocal has static local x */
function useGlobal() /* useGlobal uses global x */
function useLocal() /* useLocal reinitializes automatic local x */
function useStaticLocal() /* static local x retains its prior value */
function useGlobal() /* global x also retains its value */
printf( "\nlocal x in main is %d\n", x );
return 0; /* indicates successful termination */
} /* end main */

/* useLocal reinitializes local variable x during each call */
function useLocal( void )
{
    int x = 25; /* initialized each time useLocal is called */
    printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
    x++;
    printf( "local x in useLocal is %d before exiting useLocal\n", x );
} /* end function useLocal */

/* useStaticLocal initializes static local variable x only the first time
the function is called; value of x is saved between calls to this function */
function useStaticLocal( void )
{
    /* initialized only first time useStaticLocal is called */
    static int x = 50;
    printf( "\nlocal static x is %d on entering useStaticLocal\n", x );
    x++;
    printf( "local static x is %d on exiting useStaticLocal\n", x );
} /* end function useStaticLocal */

/* function useGlobal modifies global variable x during each call */
function useGlobal( void )
{
    printf( "\nglobal x is %d on entering useGlobal\n", x );
    x *= 10;
    printf( "global x is %d on exiting useGlobal\n", x );
} /* end function useGlobal */

local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5
local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal
local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

Fig. 5.12 | Scoping example. (Part 2 of 3.)
5.14 Recursion

The programs we’ve discussed are generally structured as functions that call one another in a disciplined, hierarchical manner. For some types of problems, it’s useful to have functions call themselves. A recursive function is a function that calls itself either directly or indirectly through another function. Recursion is a complex topic discussed at length in upper-level computer science courses. In this section and the next, simple examples of recursion are presented. This book contains an extensive treatment of recursion, which is spread throughout Chapters 5–8, 12 and Appendix F. Figure 5.17, in Section 5.16, summarizes the 31 recursion examples and exercises in the book.

We consider recursion conceptually first, and then examine several programs containing recursive functions. Recursive problem-solving approaches have a number of elements in common. A recursive function is called to solve a problem. The function actually knows how to solve only the simplest case(s), or so-called base case(s). If the function is called with a base case, the function simply returns a result. If the function is called with a more complex problem, the function divides the problem into two conceptual pieces: a piece that the function knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version. Because this new problem looks like the original problem, the function launches (calls) a fresh copy of itself to go to work on the smaller problem—this is referred to as a recursive call and is also called the recursion step. The recursion step also includes the keyword return, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller, possibly main.

The recursion step executes while the original call to the function is still open, i.e., it has not yet finished executing. The recursion step can result in many more such recursive calls, as the function keeps dividing each problem it’s called with into two conceptual pieces. In order for the recursion to terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller problems must eventually converge on the base case. At that point, the function recognizes the base case, returns a result to the previous copy of the function, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result.
to main. All of this sounds quite exotic compared to the kind of problem solving we’ve been using with conventional function calls to this point. Indeed, it takes a great deal of practice writing recursive programs before the process will appear natural. As an example of these concepts at work, let’s write a recursive program to perform a popular mathematical calculation.

**Recursively Calculating Factorials**

The factorial of a nonnegative integer \( n \), written \( n! \) (and pronounced “\( n \) factorial”), is the product

\[
  n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1
\]

with \( 1! \) equal to 1, and \( 0! \) defined to be 1. For example, \( 5! \) is the product \( 5 \times 4 \times 3 \times 2 \times 1 \), which is equal to 120.

The factorial of an integer, \( \text{number} \), greater than or equal to 0 can be calculated iteratively (nonrecursively) using a `for` statement as follows:

```c
factorial = 1;
for (counter = number; counter >= 1; counter--)
    factorial *= counter;
```

A recursive definition of the factorial function is arrived at by observing the following relationship:

\[
  n! = n \cdot (n-1)!
\]

For example, \( 5! \) is clearly equal to \( 5 \times 4! \) as is shown by the following:

\[
  5! = 5 \times 4 \times 3 \times 2 \times 1 \\
  5! = 5 \times (4 \times 3 \times 2 \times 1) \\
  5! = 5 \times (4!)
\]

The evaluation of \( 5! \) would proceed as shown in Fig. 5.13. Figure 5.13(a) shows how the succession of recursive calls proceeds until \( 1! \) is evaluated to be 1, which terminates the recursion. Figure 5.13(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.

Figure 5.14 uses recursion to calculate and print the factorials of the integers 0–10 (the choice of the type `long` will be explained momentarily). The recursive `factorial` function first tests whether a terminating condition is true, i.e., whether \( \text{number} \) is less than or equal to 1. If \( \text{number} \) is indeed less than or equal to 1, `factorial` returns 1, no further recursion is necessary, and the program terminates. If \( \text{number} \) is greater than 1, the statement

```c
return number * factorial(number - 1);
```

expresses the problem as the product of \( \text{number} \) and a recursive call to `factorial` evaluating the factorial of \( \text{number} - 1 \). The call `factorial(number - 1)` is a slightly simpler problem than the original calculation `factorial(number)`.

Function `factorial` (line 22) has been declared to receive a parameter of type `long` and return a result of type `long`. This is shorthand notation for `long int`. The C standard specifies that a variable of type `long int` is stored in at least 4 bytes, and thus may hold a value as large as \(+2147483647\). As can be seen in Fig. 5.14, factorial values become large quickly. We’ve chosen the data type `long` so the program can calculate factorials greater
5.14 Recursion

Fig. 5.13 | Recursive evaluation of 5!.

```
/* Fig. 5.14: fig05_14.c */
#include <stdio.h>

long factorial( long number ); /* function prototype */

/* function main begins program execution */
int main( void )
{
    int i; /* counter */

    /* loop 11 times; during each iteration, calculate factorial( i ) and display result */
    for ( i = 0; i <= 10; i++ ) {
        printf( "%2d! = %ld\n", i, factorial( i ) );
    } /* end for */

    return 0; /* indicates successful termination */
} /* end main */

/* recursive definition of function factorial */
long factorial( long number )
{
    /* base case */
    if ( number <= 1 ) {
        return 1;
    } /* end if */
```

Fig. 5.14 | Calculating factorials with a recursive function. (Part 1 of 2.)
Chapter 5  C Functions

than 7! on computers with small (such as 2-byte) integers. The conversion specifier %ld is used to print long values. Unfortunately, the factorial function produces large values so quickly that even long int does not help us print many factorial values before the size of a long int variable is exceeded.

As we’ll explore in the exercises, double may ultimately be needed by the user desiring to calculate factorials of larger numbers. This points to a weakness in C (and most other procedural programming languages), namely that the language is not easily extended to handle the unique requirements of various applications. As we’ll see later in the book, C++ is an extensible language that, through “classes,” allows us to create arbitrarily large integers if we wish.

**Common Programming Error 5.13**
Forgetting to return a value from a recursive function when one is needed.

**Common Programming Error 5.14**
Either omitting the base case, or writing the recursion step incorrectly so that it does not converge on the base case, will cause infinite recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution. Infinite recursion can also be caused by providing an unexpected input.

### 5.15 Example Using Recursion: Fibonacci Series

The Fibonacci series

\[0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\]

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

The series occurs in nature and, in particular, describes a form of spiral. The ratio of successive Fibonacci numbers converges to a constant value of 1.618… . This number, too,
5.15 Example Using Recursion: Fibonacci Series

repeatedly occurs in nature and has been called the golden ratio or the golden mean. Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms, and buildings whose length and width are in the ratio of the golden mean. Postcards are often designed with a golden mean length/width ratio.

The Fibonacci series may be defined recursively as follows:

\[
\begin{align*}
\text{fibonacci}(0) &= 0 \\
\text{fibonacci}(1) &= 1 \\
\text{fibonacci}(n) &= \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)
\end{align*}
\]

Figure 5.15 calculates the \(n\)th Fibonacci number recursively using function \text{fibonacci}. Notice that Fibonacci numbers tend to become large quickly. Therefore, we’ve chosen the data type \text{long} for the parameter type and the return type in function \text{fibonacci}. In Fig. 5.15, each pair of output lines shows a separate run of the program.

```c
/* Fig. 5.15: fig05_15.c */
#include <stdio.h>

long fibonacci(long n); /* function prototype */

int main(void)
{
    long result; /* fibonacci value */
    long number; /* number input by user */

    /* obtain integer from user */
    printf("Enter an integer: ");
    scanf("%ld", &number);

    /* calculate fibonacci value for number input by user */
    result = fibonacci(number);

    /* display result */
    printf("Fibonacci( %ld ) = %ld\n", number, result);
    return 0; /* indicates successful termination */
} /* end main */

/* Recursive definition of function fibonacci */
long fibonacci(long n)
{
    /* base case */
    if ( n == 0 || n == 1 ) {
        return n;
    } /* end if */
    else { /* recursive step */
        return fibonacci( n - 1 ) + fibonacci( n - 2 );
    } /* end else */
} /* end function fibonacci */
```

Fig. 5.15 | Recursively generating Fibonacci numbers. (Part 1 of 2.)
The call to `fibonacci` from `main` is not a recursive call (line 18), but all subsequent calls to `fibonacci` are recursive (line 33). Each time `fibonacci` is invoked, it immediately tests for the base case—\( n \) is equal to 0 or 1. If this is true, \( n \) is returned. Interestingly, if \( n \) is greater than 1, the recursion step generates *two* recursive calls, each of which is for a slightly simpler problem than the original call to `fibonacci`. Figure 5.16 shows how function `fibonacci` would evaluate `fibonacci(3)`.

![Fig. 5.15](image-url)  
*Recursively generating Fibonacci numbers. (Part 2 of 2.)*
Order of Evaluation of Operands

This figure raises some interesting issues about the order in which C compilers will evaluate the operands of operators. This is a different issue from the order in which operators are applied to their operands, namely the order dictated by the rules of operator precedence. From Fig. 5.16 it appears that while evaluating \texttt{fibonacci(3)} , two recursive calls will be made, namely \texttt{fibonacci(2)} and \texttt{fibonacci(1)} . But in what order will these calls be made? Most programmers simply assume the operands will be evaluated left to right. Strangely, Standard C does not specify the order in which the operands of most operators (including +) are to be evaluated. Therefore, you may make no assumption about the order in which these calls will execute. The calls could in fact execute \texttt{fibonacci(2)} first and then \texttt{fibonacci(1)} , or the calls could execute in the reverse order, \texttt{fibonacci(1)} then \texttt{fibonacci(2)} . In this program and in most other programs, it turns out the final result would be the same. But in some programs the evaluation of an operand may have side effects that could affect the final result of the expression. Of C’s many operators, Standard C specifies the order of evaluation of the operands of only four operators—namely &&, ||, the comma (, ) operator and ?: . The first three of these are binary operators whose two operands are guaranteed to be evaluated left to right. [Note: The commas used to separate the arguments in a function call are not comma operators.] The last operator is C’s only ternary operator. Its leftmost operand is always evaluated first; if the leftmost operand evaluates to nonzero, the middle operand is evaluated next and the last operand is ignored; if the leftmost operand evaluates to zero, the third operand is evaluated next and the middle operand is ignored.

Common Programming Error 5.15

Writing programs that depend on the order of evaluation of the operands of operators other than &&, ||, ?, and the comma (, ) operator can lead to errors because compilers may not necessarily evaluate the operands in the order you expect.

Fig. 5.16 | Set of recursive calls for \texttt{fibonacci(3)} .
Exponential Complexity
A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each level of recursion in the `fibonacci` function has a doubling effect on the number of calls; i.e., the number of recursive calls that will be executed to calculate the \(n\)th Fibonacci number is on the order of \(2^n\). This rapidly gets out of hand. Calculating only the 20th Fibonacci number would require on the order of \(2^{20}\) or about a million calls, calculating the 30th Fibonacci number would require on the order of \(2^{30}\) or about a billion calls, and so on. Computer scientists refer to this as exponential complexity. Problems of this nature humble even the world’s most powerful computers! Complexity issues in general, and exponential complexity in particular, are discussed in detail in the upper-level computer science curriculum course generally called “Algorithms.”

The example we showed in this section used an intuitively appealing solution to calculate Fibonacci numbers, but there are better approaches. Exercise 5.48 asks you to investigate recursion in more depth and propose alternate approaches to implementing the recursive Fibonacci algorithm.

5.16 Recursion vs. Iteration
In the previous sections, we studied two functions that can easily be implemented either recursively or iteratively. In this section, we compare the two approaches and discuss why you might choose one approach over the other in a particular situation.

Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure. Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated function calls. Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized. Iteration with counter-controlled repetition and recursion each gradually approach termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached. Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.

Recursion has many negatives. It repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space. Each recursive call causes another copy of the function (actually only the function’s variables) to be created; this can consume considerable memory. Iteration nor-
mally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted. So why choose recursion?

**Software Engineering Observation 5.13**

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution may not be apparent.

**Performance Tip 5.5**

Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

**Common Programming Error 5.16**

Accidentally having a nonrecursive function call itself either directly, or indirectly through another function.

Most programming textbooks introduce recursion much later than we’ve done here. We feel that recursion is a sufficiently rich and complex topic that it’s better to introduce it earlier and spread the examples over the remainder of the text. Figure 5.17 summarizes by chapter the 31 recursion examples and exercises in the text.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Recursion examples and exercises</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 5</td>
<td>Factorial function</td>
</tr>
<tr>
<td></td>
<td>Fibonacci function</td>
</tr>
<tr>
<td></td>
<td>Greatest common divisor</td>
</tr>
<tr>
<td></td>
<td>Sum of two integers</td>
</tr>
<tr>
<td></td>
<td>Multiply two integers</td>
</tr>
<tr>
<td></td>
<td>Raising an integer to an integer power</td>
</tr>
<tr>
<td></td>
<td>Towers of Hanoi</td>
</tr>
<tr>
<td></td>
<td>Recursive main</td>
</tr>
<tr>
<td></td>
<td>Printing keyboard inputs in reverse</td>
</tr>
<tr>
<td></td>
<td>Visualizing recursion</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Sum the elements of an array</td>
</tr>
<tr>
<td></td>
<td>Print an array</td>
</tr>
<tr>
<td></td>
<td>Print an array backward</td>
</tr>
<tr>
<td></td>
<td>Print a string backward</td>
</tr>
<tr>
<td></td>
<td>Check if a string is a palindrome</td>
</tr>
<tr>
<td></td>
<td>Minimum value in an array</td>
</tr>
<tr>
<td></td>
<td>Linear search</td>
</tr>
<tr>
<td></td>
<td>Binary search</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>Eight Queens</td>
</tr>
<tr>
<td></td>
<td>Maze traversal</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>Printing a string input at the keyboard backward</td>
</tr>
</tbody>
</table>

**Fig. 5.17** | Recursion examples and exercises in the text. (Part 1 of 2.)
Let’s close this chapter with some observations that we make repeatedly throughout the book. Good software engineering is important. High performance is important. Unfortunately, these goals are often at odds with one another. Good software engineering is key to making more manageable the task of developing the larger and more complex software systems we need. High performance is key to realizing the systems of the future that will place ever greater computing demands on hardware. Where do functions fit in here?

Performance Tip 5.6

Functionalizing programs in a neat, hierarchical manner promotes good software engineering. But it has a price. A heavily functionalized program—as compared to a monolithic (i.e., one-piece) program without functions—makes potentially large numbers of function calls, and these consume execution time on a computer’s processor(s). So, although monolithic programs may perform better, they’re more difficult to program, test, debug, maintain, and evolve.

Summary

Section 5.1 Introduction

• The best way to develop and maintain a large program is to divide it into several smaller program modules, each of which is more manageable than the original program. Modules are written as functions in C.

Section 5.2 Program Modules in C

• A function is invoked by a function call. The function call mentions the function by name and provides information (as arguments) that the called function needs to perform its task.

• The purpose of information hiding is for functions to have access only to the information they need to complete their tasks. This is a means of implementing the principle of least privilege, one of the most important principles of good software engineering.

Section 5.3 Math Library Functions

• Functions are normally invoked in a program by writing the name of the function followed by a left parenthesis followed by the argument (or a comma-separated list of arguments) of the function followed by a right parenthesis.
• Data type `double` is a floating-point type like `float`. A variable of type `double` can store a value of much greater magnitude and precision than `float` can store.

• Each argument of a function may be a constant, a variable, or an expression.

**Section 5.4 Functions**
• A local variable is known only in a function definition. Other functions are not allowed to know the names of a function’s local variables, nor is any function allowed to know the implementation details of any other function.

**Section 5.5 Function Definitions**
• The general format for a function definition is
  
  \[
  \text{return-value-type function-name( parameter-list )} \\
  \{ \\
  \text{definitions} \\
  \text{statements} \\
  \}
  \]

  The `return-value-type` states the type of the value returned to the calling function. If a function does not return a value, the `return-value-type` is declared as `void`. The `function-name` is any valid identifier. The `parameter-list` is a comma-separated list containing the definitions of the variables that will be passed to the function. If a function does not receive any values, `parameter-list` is declared as `void`. The `function-body` is the set of definitions and statements that constitute the function.

• The arguments passed to a function should match in number, type and order with the parameters in the function definition.

• When a program encounters a function call, control is transferred from the point of invocation to the called function, the statements of the called function are executed and control returns to the caller.

• A called function can return control to the caller in one of three ways. If the function does not return a value, control is returned when the function-ending right brace is reached, or by executing the statement
  
  \[
  \text{return} ;
  \]

  If the function does return a value, the statement
  
  \[
  \text{return expression} ;
  \]

  returns the value of `expression`.

**Section 5.6 Function Prototypes**
• A function prototype declares the return type of the function and declares the number, the types, and order of the parameters the function expects to receive.

• Function prototypes enable the compiler to verify that functions are called correctly.

• The compiler ignores variable names mentioned in the function prototype.

**Section 5.7 Function Call Stack and Activation Records**
• Stacks are known as last-in, first-out (LIFO) data structures—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

• A called function must know how to return to its caller, so the return address of the calling function is pushed onto the program execution stack when the function is called. If a series of function calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order so that the last function to execute will be the first to return to its caller.
• The program execution stack contains the memory for the local variables used in each invocation of a function during a program’s execution. This data is known as the activation record or stack frame of the function call. When a function call is made, the activation record for that function call is pushed onto the program execution stack. When the function returns to its caller, the activation record for this function call is popped off the stack and those local variables are no longer known to the program.

• The amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the program execution stack. If there are more function calls than can have their activation records stored on the program execution stack, an error known as a stack overflow occurs. The application will compile correctly, but its execution causes a stack overflow.

**Section 5.8 Headers**

• Each standard library has a corresponding header containing the function prototypes for all the functions in that library, as well as definitions of various symbolic constants needed by those functions.

• You can create and include your own headers.

**Section 5.9 Calling Functions By Value and By Reference**

• When an argument is passed by value, a copy of the variable’s value is made and the copy is passed to the called function. Changes to the copy in the called function do not affect the original variable’s value.

• All calls in C are call-by-value.

• It’s possible to simulate call-by-reference by using address operators and indirection operators.

**Section 5.10 Random Number Generation**

• Function `rand` generates an integer between 0 and `RAND_MAX` which is defined by the C standard to be at least 32767.

• The function prototypes for `rand` and `srand` are contained in `<stdlib.h>`.

• Values produced by `rand` can be scaled and shifted to produce values in a specific range.

• To randomize a program, use the C Standard Library function `srand`.

• The `srand` function call is ordinarily inserted in a program only after it has been thoroughly debugged. While debugging, it’s better to omit `srand`. This ensures repeatability, which is essential to proving that corrections to a random number generation program work properly.

• To randomize without the need for entering a seed each time, we use `srand(time(NULL))`.

• The general equation for scaling and shifting a random number is

\[ n = a + \text{rand()} \% b; \]

where \( a \) is the shifting value (i.e., the first number in the desired range of consecutive integers) and \( b \) is the scaling factor (i.e., the width of the desired range of consecutive integers).

**Section 5.11 Example: A Game of Chance**

• An enumeration, introduced by the keyword `enum`, is a set of integer constants represented by identifiers. Values in an `enum` start with 0 and are incremented by 1. It’s also possible to assign an integer value to each identifier in an `enum`. The identifiers in an enumeration must be unique, but the values may be duplicated.

**Section 5.12 Storage Classes**

• Each identifier in a program has the attributes storage class, storage duration, scope and linkage.
• C provides four storage classes indicated by the storage class specifiers: auto, register, extern and static; only one storage class specifier can be used for a given declaration.

• An identifier’s storage duration is when that identifier exists in memory.

**Section 5.13 Scope Rules**

• An identifier’s scope is where the identifier can be referenced in a program.

• An identifier’s linkage determines for a multiple-source-file program whether an identifier is known only in the current source file or in any source file with proper declarations.

• Variables with automatic storage duration are created when the block in which they’re defined is entered, exist while the block is active and are destroyed when the block is exited. A function’s local variables normally have automatic storage duration.

• The storage class specifier register can be placed before an automatic variable declaration to suggest that the compiler maintain the variable in one of the computer’s high-speed hardware registers. The compiler may ignore register declarations. Keyword register can be used only with variables of automatic storage duration.

• Keywords extern and static are used to declare identifiers for variables and functions of static storage duration.

• Variables with static storage duration are allocated and initialized once, when the program begins execution.

• There are two types of identifiers with static storage duration: external identifiers (such as global variables and function names) and local variables declared with the storage-class specifier static.

• Global variables are created by placing variable definitions outside any function definition. Global variables retain their values throughout the execution of the program.

• Local variables declared static retain their value between calls to the function in which they’re defined.

• All numeric variables of static storage duration are initialized to zero if you do not explicitly initialize them.

• The four scopes for an identifier are function scope, file scope, block scope and function-proto-type scope.

• Labels are the only identifiers with function scope. Labels can be used anywhere in the function in which they appear but cannot be referenced outside the function body.

• An identifier declared outside any function has file scope. Such an identifier is “known” in all functions from the point at which the identifier is declared until the end of the file.

• Identifiers defined inside a block have block scope. Block scope ends at the terminating right brace (}) of the block.

• Local variables defined at the beginning of a function have block scope, as do function parameters, which are considered local variables by the function.

• Any block may contain variable definitions. When blocks are nested, and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is “hidden” until the inner block terminates.

• The only identifiers with function-proto-type scope are those used in the parameter list of a function prototype. Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity.

**Section 5.14 Recursion**

• A recursive function is a function that calls itself either directly or indirectly.
• If a recursive function is called with a base case, the function simply returns a result. If the function is called with a more complex problem, the function divides the problem into two conceptual pieces: a piece that the function knows how to do and a slightly smaller version of the original problem. Because this new problem looks like the original problem, the function launches a recursive call to work on the smaller problem.

• For recursion to terminate, each time the recursive function calls itself with a slightly simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case. When the function recognizes the base case, the result is returned to the previous function call, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result.

• Standard C does not specify the order in which the operands of most operators (including +) are to be evaluated. Of C’s many operators, the standard specifies the order of evaluation of the operands of only the operators &&, ||, the comma (,) operator and ?. The first three of these are binary operators whose two operands are evaluated left to right. The last operator is C’s only ternary operator. Its leftmost operand is evaluated first; if the leftmost operand evaluates to nonzero, the middle operand is evaluated next and the last operand is ignored; if the leftmost operand evaluates to zero, the third operand is evaluated next and the middle operand is ignored.

Section 5.16 Recursion vs. Iteration

• Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure.

• Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated function calls.

• Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.

• Iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base case.

• Recursion repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space.

Terminology

abstraction 144  
activation record 151  
argument (of a function) 142  
auto 161  
automatic storage duration 162  
automatic variable 162  
base case 167  
block 146  
block scope 164  
C Standard Library 141  
call a function 142  
call-by-reference 152  
call-by-value 152  
called 145  
called function 142  
caller 142  
calling function 142  
coercion of arguments 149  
divide and conquer 141  
enum 161  
enumeration 161  
enumeration constant 161  
file scope 164  
function 141  
function body 146  
function call 142  
function call stack 151  
function prototype 145  
function-prototype scope 164  
function scope 164  
header 146  
information hiding 164  
invoke a function 142  
invoked 142
Self-Review Exercises

5.1 Answer each of the following:

a) A program module in C is called a(n) _______.

b) A function is invoked with a(n) _______.

c) A variable that is known only within the function in which it's defined is called a(n) _______.

d) The _______ statement in a called function is used to pass the value of an expression back to the calling function.

e) Keyword _______ is used in a function header to indicate that a function does not return a value or to indicate that a function contains no parameters.

f) The _______ of an identifier is the portion of the program in which the identifier can be used.

g) The three ways to return control from a called function to a caller are _______, _______ and _______.

h) A(n) _______ allows the compiler to check the number, types, and order of the arguments passed to a function.

i) The _______ function is used to produce random numbers.

j) The _______ function is used to set the random number seed to randomize a program.

k) The storage-class specifiers are _______, _______, _______ and _______.

l) Variables declared in a block or in the parameter list of a function are assumed to be of storage class _______ unless specified otherwise.

m) The storage-class specifier _______ is a recommendation to the compiler to store a variable in one of the computer's registers.

n) A non-static variable defined outside any block or function is a(n) _______ variable.

o) For a local variable in a function to retain its value between calls to the function, it must be declared with the _______ storage-class specifier.

p) The four possible scopes of an identifier are _______, _______, _______ and _______.

last-in-first-out (LIFO) 151
linkage 161
linkage of an identifier 162
local variable 144
mixed-type expression 150
module 141
parameter 144
parameter-list 146
pop off a stack 151
principle of least privilege 164
program execution stack 151
programmer-defined function 142
promotion rule 149
pseudorandom numbers 156
push onto a stack 151
randomizing 156
recursion step 167
recursive call 167
recursive function 167
return from a function 142
return value type 146

scaling 153
scaling factor 153
scope 162
scope of an identifier 164
seed the rand function 156
shift 153
shifting value 158
side effect 152
simulate 152
software reusability 144
stack 151
stack frame 151
stack overflow 151
standard library header 151
static 161
static storage duration 162
storage class 161
storage class of an identifier 161
storage class specifier 161
storage duration 161
q) A function that calls itself either directly or indirectly is a(n) ______ function.
r) A recursive function typically has two components: one that provides a means for the recursion to terminate by testing for a(n) ______ case, and one that expresses the problem as a recursive call for a slightly simpler problem than the original call.

5.2 For the following program, state the scope (either function scope, file scope, block scope or function prototype scope) of each of the following elements.
   a) The variable \( x \) in \( \text{main} \).
   b) The variable \( y \) in \( \text{cube} \).
   c) The function \( \text{cube} \).
   d) The function \( \text{main} \).
   e) The function prototype for \( \text{cube} \).
   f) The identifier \( y \) in the function prototype for \( \text{cube} \).

```c
#include <stdio.h>

int cube( int y );

int main( void )
{
    int x;

    for ( x = 1; x <= 10; x++ )
        printf( "%d\n", cube( x ) );

    return 0;
}

int cube( int y )
{
    return y * y * y;
}
```

5.3 Write a program that tests whether the examples of the math library function calls shown in Fig. 5.2 actually produce the indicated results.

5.4 Give the function header for each of the following functions.
   a) Function \( \text{hypotenuse} \) that takes two double-precision floating-point arguments, \( \text{side1} \) and \( \text{side2} \), and returns a double-precision floating-point result.
   b) Function \( \text{smallest} \) that takes three integers, \( x \), \( y \), \( z \), and returns an integer.
   c) Function \( \text{instructions} \) that does not receive any arguments and does not return a value. [Note: Such functions are commonly used to display instructions to a user.]
   d) Function \( \text{intToFloat} \) that takes an integer argument, \( \text{number} \), and returns a floating-point result.

5.5 Give the function prototype for each of the following:
   a) The function described in Exercise 5.4(a).
   b) The function described in Exercise 5.4(b).
   c) The function described in Exercise 5.4(c).
   d) The function described in Exercise 5.4(d).

5.6 Write a declaration for each of the following:
   a) Integer \( \text{count} \) that should be maintained in a register. Initialize \( \text{count} \) to 0.
   b) Floating-point variable \( \text{lastVal} \) that is to retain its value between calls to the function in which it’s defined.
   c) External integer \( \text{number} \) whose scope should be restricted to the remainder of the file in which it’s defined.
5.7 Find the error in each of the following program segments and explain how the error can be corrected (see also Exercise 5.46):

a) int g( void )
   {
     printf( "Inside function g\n" );
     int h( void )
     {
       printf( "Inside function h\n" );
     }
   }

b) int sum( int x, int y )
   {
     int result;
     result = x + y;
   }

c) int sum( int n )
   {
     if ( n == 0 ) {
       return 0;
     }
     else {
       n + sum( n - 1 );
     }
   }

d) void f( float a );
   {
     float a;
     printf( "%f", a );
   }

e) void product( void )
   {
     int a, b, c, result;
     printf( "Enter three integers: " )
     scanf( "%d%d%d", &a, &b, &c );
     result = a * b * c;
     printf( "Result is %d", result );
     return result;
   }

Answers to Self-Review Exercises

5.1 a) Function. b) Function call. c) Local variable. d) return. e) void. f) Scope. g) return; or return expression; or encountering the closing right brace of a function. h) Function prototype. i) rand. j) srand. k) auto, register, extern, static. l) auto. m) register. n) External, global. o) static. p) Function scope, file scope, block scope, function prototype scope. q) Recursive. r) Base.

/* ex05_03.c */
/* Testing the math library functions */
#include <stdio.h>
#include <math.h>

/* function main begins program execution */
int main(void)
{
    /* calculates and outputs the square root */
    printf("sqrt(%.1f) = %.1f\n", 900.0, sqrt(900.0));
    printf("sqrt(%.1f) = %.1f\n", 9.0, sqrt(9.0));

    /* calculates and outputs the exponential function e to the x */
    printf("exp(%.1f) = %f\n", 1.0, exp(1.0));
    printf("exp(%.1f) = %f\n", 2.0, exp(2.0));

    /* calculates and outputs the logarithm (base e) */
    printf("log(%f) = %.1f\n", 2.718282, log(2.718282));
    printf("log(%f) = %.1f\n", 7.389056, log(7.389056));

    /* calculates and outputs the logarithm (base 10) */
    printf("log10(%.1f) = %.1f\n", 1.0, log10(1.0));
    printf("log10(%.1f) = %.1f\n", 10.0, log10(10.0));
    printf("log10(%.1f) = %.1f\n", 100.0, log10(100.0));

    /* calculates and outputs the absolute value */
    printf("fabs(%.1f) = %.1f\n", 13.5, fabs(13.5));
    printf("fabs(%.1f) = %.1f\n", 0.0, fabs(0.0));
    printf("fabs(%.1f) = %.1f\n", -13.5, fabs(-13.5));

    /* calculates and outputs ceil( x ) */
    printf("ceil(%.1f) = %.1f\n", 9.2, ceil(9.2));
    printf("ceil(%.1f) = %.1f\n", -9.8, ceil(-9.8));

    /* calculates and outputs floor( x ) */
    printf("floor(%.1f) = %.1f\n", 9.2, floor(9.2));
    printf("floor(%.1f) = %.1f\n", -9.8, floor(-9.8));

    /* calculates and outputs pow( x, y ) */
    printf("pow(%.1f, %.1f) = %.1f\n", 2.0, 7.0, pow(2.0, 7.0));
    printf("pow(%.1f, %.1f) = %.1f\n", 9.0, 0.5, pow(9.0, 0.5));

    /* calculates and outputs fmod( x, y ) */
    printf("fmod(%.3f/%.3f) = %.3f\n", 13.675, 2.333, fmod(13.675, 2.333));

    /* calculates and outputs sin( x ) */
    printf("sin(%.1f) = %.1f\n", 0.0, sin(0.0));

    /* calculates and outputs cos( x ) */
    printf("cos(%.1f) = %.1f\n", 0.0, cos(0.0));

    /* calculates and outputs tan( x ) */
    printf("tan(%.1f) = %.1f\n", 0.0, tan(0.0));

    return 0; /* indicates successful termination */
} /* end main */
5.4  a)  \texttt{double hypotenuse( double side1, double side2 )}
    b)  \texttt{int smallest( int x, int y, int z )}
    c)  \texttt{void instructions( void )}
    d)  \texttt{float intToFloat( int number )}

5.5  a)  \texttt{double hypotenuse( double side1, double side2 );}
    b)  \texttt{int smallest( int x, int y, int z );}
    c)  \texttt{void instructions( void );}
    d)  \texttt{float intToFloat( int number );}

5.6  a)  \texttt{register int count = 0;}
    b)  \texttt{static float lastVal;}
    c)  \texttt{static int number;}
       \hspace{1em} [\textit{Note: This would appear outside any function definition.}]

5.7  a)  Error: Function h is defined in function g.
    Correction: Move the definition of h out of the definition of g.

b)  Error: The body of the function is supposed to return an integer, but does not.
    Correction: Delete variable \texttt{result} and place the following statement in the function:

    \begin{verbatim}
    return x + y;
    \end{verbatim}

c)  Error: The result of \texttt{n + sum(n - 1)} is not returned; \texttt{sum} returns an improper result.
    Correction: Rewrite the statement in the \texttt{else} clause as

    \begin{verbatim}
    return n + sum(n - 1);
    \end{verbatim}

d)  Error: Semicolon after the right parenthesis that encloses the parameter list, and re-
    defining the parameter \texttt{a} in the function definition.
    Correction: Delete the semicolon after the right parenthesis of the parameter list, and
    delete the declaration \texttt{float a;} in the function body.

e)  Error: The function returns a value when it’s not supposed to.
    Correction: Eliminate the \texttt{return} statement.
Exercises

5.8 Show the value of \( x \) after each of the following statements is performed:
\[ \begin{align*}
a) & \quad x = \text{fabs}( 7.5 ); \\
b) & \quad x = \text{floor}( 7.5 ); \\
c) & \quad x = \text{fabs}( 0.0 ); \\
d) & \quad x = \text{ceil}( 0.0 ); \\
e) & \quad x = \text{fabs}( -6.4 ); \\
f) & \quad x = \text{ceil}( -6.4 ); \\
g) & \quad x = \text{ceil}( -\text{fabs}( -8 + \text{floor}( -5.5 ) ) ); \\
\end{align*} \]

5.9 (Parking Charges) A parking garage charges a $2.00 minimum fee to park for up to three hours and an additional $0.50 per hour for each hour or part thereof over three hours. The maximum charge for any given 24-hour period is $10.00. Assume that no car parks for longer than 24 hours at a time. Write a program that will calculate and print the parking charges for each of three customers who parked their cars in this garage yesterday. You should enter the hours parked for each customer. Your program should print the results in a neat tabular format, and should calculate and print the total of yesterday’s receipts. The program should use the function `calculateCharges` to determine the charge for each customer. Your outputs should appear in the following format:

<table>
<thead>
<tr>
<th>Car</th>
<th>Hours</th>
<th>Charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.5</td>
<td>2.00</td>
</tr>
<tr>
<td>2</td>
<td>4.0</td>
<td>2.50</td>
</tr>
<tr>
<td>3</td>
<td>24.0</td>
<td>10.00</td>
</tr>
<tr>
<td>TOTAL</td>
<td>29.5</td>
<td>14.50</td>
</tr>
</tbody>
</table>

5.10 (Rounding Numbers) An application of function `floor` is rounding a value to the nearest integer. The statement
\[ y = \text{floor}( x + 0.5 ); \]
will round the number \( x \) to the nearest integer and assign the result to \( y \). Write a program that reads several numbers and uses the preceding statement to round each of these numbers to the nearest integer. For each number processed, print both the original number and the rounded number.

5.11 (Rounding Numbers) Function `floor` may be used to round a number to a specific decimal place. The statement
\[ y = \text{floor}( x * 10 + 0.5 ) / 10; \]
rounds \( x \) to the tenths position (the first position to the right of the decimal point). The statement
\[ y = \text{floor}( x * 100 + 0.5 ) / 100; \]
rounds \( x \) to the hundredths position (the second position to the right of the decimal point). Write a program that defines four functions to round a number \( x \) in various ways
\[ \begin{align*}
a) & \quad \text{roundToInteger( number )} \\
b) & \quad \text{roundToTenths( number )} \\
c) & \quad \text{roundToHundredths( number )} \\
d) & \quad \text{roundToThousandths( number )} \\
\end{align*} \]
For each value read, your program should print the original value, the number rounded to the nearest integer, the number rounded to the nearest tenth, the number rounded to the nearest hundredth, and the number rounded to the nearest thousandth.

5.12 Answer each of the following questions.
\[ \begin{align*}
a) & \quad \text{What does it mean to choose numbers “at random”?} \\
b) & \quad \text{Why is the \text{rand} function useful for simulating games of chance?} \\
c) & \quad \text{Why would you randomize a program by using \text{srand}? Under what circumstances is it desirable not to randomize?} \\
\end{align*} \]
d) Why is it often necessary to scale and/or shift the values produced by rand?

c) Why is computerized simulation of real-world situations a useful technique?

5.13 Write statements that assign random integers to the variable \( n \) in the following ranges:

- a) \( 1 \leq n \leq 2 \)
- b) \( 1 \leq n \leq 100 \)
- c) \( 0 \leq n \leq 9 \)
- d) \( 1000 \leq n \leq 1112 \)
- e) \( -1 \leq n \leq 1 \)
- f) \( -3 \leq n \leq 11 \)

5.14 For each of the following sets of integers, write a single statement that will print a number at random from the set.

- a) 2, 4, 6, 8, 10.
- b) 3, 5, 7, 9, 11.
- c) 6, 10, 14, 18, 22.

5.15 (Hypotenuse Calculations) Define a function called hypotenuse that calculates the length of the hypotenuse of a right triangle when the other two sides are given. Use this function in a program to determine the length of the hypotenuse for each of the following triangles. The function should take two arguments of type double and return the hypotenuse as a double. Test your program with the side values specified in Fig. 5.18.

5.16 (Exponentiation) Write a function integerPower(base, exponent) that returns the value of \( base^{exponent} \)

For example, \( integerPower(3, 4) = 3 * 3 * 3 * 3 \). Assume that exponent is a positive, nonzero integer, and base is an integer. Function integerPower should use for to control the calculation. Do not use any math library functions.

5.17 (Multiples) Write a function multiple that determines for a pair of integers whether the second integer is a multiple of the first. The function should take two integer arguments and return 1 (true) if the second is a multiple of the first, and 0 (false) otherwise. Use this function in a program that inputs a series of pairs of integers.

5.18 (Even or Odd) Write a program that inputs a series of integers and passes them one at a time to function even, which uses the remainder operator to determine if an integer is even. The function should take an integer argument and return 1 if the integer is even and 0 otherwise.

5.19 (Parking Charges) Write a function that displays a solid square of asterisks whose side is specified in integer parameter side. For example, if side is 4, the function displays:

```
****
****
****
****
```

5.20 (Displaying a Square of Any Character) Modify the function created in Exercise 5.19 to form the square out of whatever character is contained in character parameter fillCharacter. Thus if side is 5 and fillCharacter is “#” then this function should print:

```
#####
#####
#####
#####
#####
```
Chapter 5  C Functions

5.21 (Project: Drawing Shapes with Characters) Use techniques similar to those developed in Exercises 5.19–5.20 to produce a program that graphs a wide range of shapes.

5.22 (Separating Digits) Write program segments that accomplish each of the following:
   a) Calculate the integer part of the quotient when integer \( a \) is divided by integer \( b \).
   b) Calculate the integer remainder when integer \( a \) is divided by integer \( b \).
   c) Use the program pieces developed in a) and b) to write a function that inputs an integer between 1 and 32767 and prints it as a series of digits, with two spaces between each digit. For example, the integer 4562 should be printed as:

   4  5  6  2

5.23 (Time in Seconds) Write a function that takes the time as three integer arguments (for hours, minutes, and seconds) and returns the number of seconds since the last time the clock “struck 12.” Use this function to calculate the amount of time in seconds between two times, both of which are within one 12-hour cycle of the clock.

5.24 (Temperature Conversions) Implement the following integer functions:
   a) Function \( celsius \) returns the Celsius equivalent of a Fahrenheit temperature.
   b) Function \( fahrenheit \) returns the Fahrenheit equivalent of a Celsius temperature.
   c) Use these functions to write a program that prints charts showing the Fahrenheit equivalents of all Celsius temperatures from 0 to 100 degrees, and the Celsius equivalents of all Fahrenheit temperatures from 32 to 212 degrees. Print the outputs in a neat tabular format that minimizes the number of lines of output while remaining readable.

5.25 (Find the Minimum) Write a function that returns the smallest of three floating-point numbers.

5.26 (Perfect Numbers) An integer number is said to be a perfect number if its factors, including 1 (but not the number itself), sum to the number. For example, 6 is a perfect number because \( 6 = 1 + 2 + 3 \). Write a function \( perfect \) that determines if parameter \( number \) is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the power of your computer by testing numbers much larger than 1000.

5.27 (Prime Numbers) An integer is said to be prime if it’s divisible by only 1 and itself. For example, 2, 3, 5 and 7 are prime, but 4, 6, 8 and 9 are not.
   a) Write a function that determines if a number is prime.
   b) Use this function in a program that determines and prints all the prime numbers between 1 and 10,000. How many of these 10,000 numbers do you really have to test before being sure that you have found all the primes?
   c) Initially you might think that \( n/2 \) is the upper limit for which you must test to see if a number is prime, but you need go only as high as the square root of \( n \). Why? Rewrite the program, and run it both ways. Estimate the performance improvement.

\[ \begin{array}{|c|c|c|}
\hline
\text{Triangle} & \text{Side 1} & \text{Side 2} \\
\hline
1 & 3.0 & 4.0 \\
2 & 5.0 & 12.0 \\
3 & 8.0 & 15.0 \\
\hline
\end{array} \]

Fig. 5.18 | Sample triangle side values for Exercise 5.15.
5.28 **(Reversing Digits)** Write a function that takes an integer value and returns the number with its digits reversed. For example, given the number 7631, the function should return 1367.

5.29 **(Greatest Common Divisor)** The greatest common divisor (GCD) of two integers is the largest integer that evenly divides each of the two numbers. Write function `gcd` that returns the greatest common divisor of two integers.

5.30 Write a function `qualityPoints` that inputs a student’s average and returns 4 if a student’s average is 90–100, 3 if the average is 80–89, 2 if the average is 70–79, 1 if the average is 60–69, and 0 if the average is lower than 60.

5.31 **(Coin Tossing)** Write a program that simulates coin tossing. For each toss of the coin the program should print Heads or Tails. Let the program toss the coin 100 times, and count the number of times each side of the coin appears. Print the results. The program should call a separate function `flip` that takes no arguments and returns 0 for tails and 1 for heads. [Note: If the program realistically simulates the coin tossing, then each side of the coin should appear approximately half the time for a total of approximately 50 heads and 50 tails.]

5.32 **(Guess the Number)** Write a C program that plays the game of “guess the number” as follows: Your program chooses the number to be guessed by selecting an integer at random in the range 1 to 1000. The program then types:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
```

The player then types a first guess. The program responds with one of the following:

```
1. Excellent! You guessed the number!
   Would you like to play again (y or n)?
2. Too low. Try again.
3. Too high. Try again.
```

If the player’s guess is incorrect, your program should loop until the player finally gets the number right. Your program should keep telling the player Too high or Too low to help the player “zero in” on the correct answer. [Note: The searching technique employed in this problem is called binary search. We’ll say more about this in the next problem.]

5.33 **(Guess the Number Modification)** Modify the program of Exercise 5.32 to count the number of guesses the player makes. If the number is 10 or fewer, print Either you know the secret or you got lucky! If the player guesses the number in 10 tries, then print Ahah! You know the secret! If the player makes more than 10 guesses, then print You should be able to do better! Why should it take no more than 10 guesses? Well, with each “good guess” the player should be able to eliminate half of the numbers. Now show why any number 1 to 1000 can be guessed in 10 or fewer tries.

5.34 **(Recursive Exponentiation)** Write a recursive function `power(base, exponent)` that when invoked returns

```
base^{exponent}
```

For example, `power(3, 4) = 3 * 3 * 3 * 3`. Assume that `exponent` is an integer greater than or equal to 1. *Hint:* The recursion step would use the relationship

```
base^{exponent} = base * base^{exponent-1}
```

and the terminating condition occurs when `exponent` is equal to 1 because

```
base^{1} = base
```
5.35 (Fibonacci) The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

begins with the terms 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms. a) Write a nonrecursive function fibonacci(n) that calculates the n\textsuperscript{th} Fibonacci number. b) Determine the largest Fibonacci number that can be printed on your system. Modify the program of part a) to use double instead of int to calculate and return Fibonacci numbers. Let the program loop until it fails because of an excessively high value.

5.36 (Towers of Hanoi) Every budding computer scientist must grapple with certain classic problems, and the Towers of Hanoi (see Fig. 5.19) is one of the most famous of these. Legend has it that in a temple in the Far East, priests are attempting to move a stack of disks from one peg to another. The initial stack had 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg under the constraints that exactly one disk is moved at a time, and at no time may a larger disk be placed above a smaller disk. A third peg is available for temporarily holding the disks. Supposedly the world will end when the priests complete their task, so there is little incentive for us to facilitate their efforts.

Let’s assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that will print the precise sequence of disk-to-disk peg transfers.

If we were to approach this problem with conventional methods, we’d rapidly find ourselves hopelessly knotted up in managing the disks. Instead, if we attack the problem with recursion in mind, it immediately becomes tractable. Moving n disks can be viewed in terms of moving only n\textendash1 disks (and hence the recursion) as follows:

a) Move n\textendash1 disks from peg 1 to peg 2, using peg 3 as a temporary holding area.

b) Move the last disk (the largest) from peg 1 to peg 3.

c) Move the n\textendash1 disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving n = 1 disk, i.e., the base case. This is accomplished by trivially moving the disk without the need for a temporary holding area.

Write a program to solve the Towers of Hanoi problem. Use a recursive function with four parameters:

a) The number of disks to be moved

b) The peg on which these disks are initially threaded

c) The peg to which this stack of disks is to be moved

d) The peg to be used as a temporary holding area

Fig. 5.19 | Towers of Hanoi for the case with four disks.
Your program should print the precise instructions it will take to move the disks from the starting peg to the destination peg. For example, to move a stack of three disks from peg 1 to peg 3, your program should print the following series of moves:

```
1 → 3 (This means move one disk from peg 1 to peg 3.)
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3
```

5.37 *(Towers of Hanoi: Iterative Solution)* Any program that can be implemented recursively can be implemented iteratively, although sometimes with considerably more difficulty and considerably less clarity. Try writing an iterative version of the Towers of Hanoi. If you succeed, compare your iterative version with the recursive version you developed in Exercise 5.36. Investigate issues of performance, clarity, and your ability to demonstrate the correctness of the programs.

5.38 *(Visualizing Recursion)* It’s interesting to watch recursion “in action.” Modify the factorial function of Fig. 5.14 to print its local variable and recursive call parameter. For each recursive call, display the outputs on a separate line and add a level of indentation. Do your utmost to make the outputs clear, interesting and meaningful. Your goal here is to design and implement an output format that helps a person understand recursion better. You may want to add such display capabilities to the many other recursion examples and exercises throughout the text.

5.39 *(Recursive Greatest Common Divisor)* The greatest common divisor of integers \(x\) and \(y\) is the largest integer that evenly divides both \(x\) and \(y\). Write a recursive function \(\text{gcd}\) that returns the greatest common divisor of \(x\) and \(y\). The \(\text{gcd}\) of \(x\) and \(y\) is defined recursively as follows: If \(y\) is equal to 0, then \(\text{gcd}(x, y) = x\); otherwise \(\text{gcd}(x, y) = \text{gcd}(y, x \mod y)\) where \(\mod\) is the remainder operator.

5.40 *(Recursive main)* Can \texttt{main} be called recursively? Write a program containing a function \texttt{main}. Include static local variable \texttt{count} initialized to 1. Postincrement and print the value of \texttt{count} each time \texttt{main} is called. Run your program. What happens?

5.41 *(Distance Between Points)* Write function \texttt{distance} that calculates the distance between two points \((x_1, y_1)\) and \((x_2, y_2)\). All numbers and return values should be of type \texttt{double}.

5.42 What does the following program do?

```c
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    int c; /* variable to hold character input by user */
    if ( ( c = getchar() ) != EOF ) {
        main();
        printf( "%c", c );
    } /* end if */
    return 0; /* indicates successful termination */
} /* end main */
```

5.43 What does the following program do?

```c
#include <stdio.h>

int mystery( int a, int b ); /* function prototype */
```
5.44 After you determine what the program of Exercise 5.43 does, modify the program to function properly after removing the restriction of the second argument being nonnegative.

5.45 (Testing Math Library Functions) Write a program that tests as many of the math library functions in Fig. 5.2 as you can. Exercise each of these functions by having your program print out tables of return values for a diversity of argument values.

5.46 Find the error in each of the following program segments and explain how to correct it:

a) `double cube( float ); /* function prototype */
cube( float number ) /* function definition */
{
    return number * number * number;
}

b) `register auto int x = 7;`

c) `int randomNumber = srand();`

d) `double y = 123.45678;`

    `int x;`
    `x = y;`
    `printf( "%f\n", (double) x );`

c) `double square( double number )`
    {
        double number;
        return number * number;
    }

e) `int sum( int n )`
    {
        if ( n == 0 ) {
            return 0;
        }
else {
    return n + sum(n);
}

5.47 (Craps Game Modification) Modify the craps program of Fig. 5.10 to allow wagering. Package as a function the portion of the program that runs one game of craps. Initialize variable bankBalance to 1000 dollars. Prompt the player to enter a wager. Use a while loop to check that wager is less than or equal to bankBalance, and if not, prompt the user to reenter wager until a valid wager is entered. After a correct wager is entered, run one game of craps. If the player wins, increase bankBalance by wager and print the new bankBalance. If the player loses, decrease bankBalance by wager, print the new bankBalance, check if bankBalance has become zero, and if so print the message, "Sorry. You busted!" As the game progresses, print various messages to create some “chatter” such as, "Oh, you're going for broke, huh?" or "Aw cmon, take a chance!" or "You're up big. Now's the time to cash in your chips!"

5.48 (Research Project: Improving the Recursive Fibonacci Implementation) In Section 5.15, the recursive algorithm we used to calculate Fibonacci numbers was intuitively appealing. However, recall that the algorithm resulted in the exponential explosion of recursive function calls. Research the recursive Fibonacci implementation online. Study the various approaches, including the iterative version in Exercise 5.35 and versions that use only so-called “tail recursion.” Discuss the relative merits of each.

Making a Difference

5.49 (Global Warming Facts Quiz) The controversial issue of global warming has been widely publicized by the film An Inconvenient Truth, featuring former Vice President Al Gore. Mr. Gore and a U.N. network of scientists, the Intergovernmental Panel on Climate Change, shared the 2007 Nobel Peace Prize in recognition of “their efforts to build up and disseminate greater knowledge about man-made climate change.” Research both sides of the global warming issue online (you might want to search for phrases like “global warming skeptics”). Create a five-question multiple-choice quiz on global warming, each question having four possible answers (numbered 1–4). Be objective and try to fairly represent both sides of the issue. Next, write an application that administers the quiz, calculates the number of correct answers (zero through five) and returns a message to the user. If the user correctly answers five questions, print “Excellent”; if four, print “Very good”; if three or fewer, print “Time to brush up on your knowledge of global warming,” and include a list of some of the websites where you found your facts.

Computer Assisted Instruction

As computer costs decline, it becomes feasible for every student, regardless of economic circumstance, to have a computer and use it in school. This creates exciting possibilities for improving the educational experience of all students worldwide as suggested by the next five exercises. [Note: Check out initiatives such as the One Laptop Per Child Project (www.laptop.org). Also, research “green” laptops—what are some key “going green” characteristics of these devices? Look into the Electronic Product Environmental Assessment Tool (www.epeat.net) which can help you assess the “greenness” of desktops, notebooks and monitors to help you decide which products to purchase.]

5.50 (Computer-Assisted Instruction) The use of computers in education is referred to as computer-assisted instruction (CAI). Write a program that will help an elementary school student learn multiplication. Use the rand function to produce two positive one-digit integers. The program should then prompt the user with a question, such as

How much is 6 times 7?
The student then inputs the answer. Next, the program checks the student's answer. If it's correct, display the message "Very good!" and ask another multiplication question. If the answer is wrong, display the message "No. Please try again." and let the student try the same question repeatedly until the student finally gets it right. A separate function should be used to generate each new question. This function should be called once when the application begins execution and each time the user answers the question correctly.

5.51 (Computer-Assisted Instruction: Reducing Student Fatigue) One problem in CAI environments is student fatigue. This can be reduced by varying the computer's responses to hold the student's attention. Modify the program of Exercise 5.50 so that various comments are displayed for each answer as follows:

Possible responses to a correct answer:

- Very good!
- Excellent!
- Nice work!
- Keep up the good work!

Possible responses to an incorrect answer:

- No. Please try again.
- Wrong. Try once more.
- Don't give up!
- No. Keep trying.

Use random-number generation to choose a number from 1 to 4 that will be used to select one of the four appropriate responses to each correct or incorrect answer. Use a switch statement to issue the responses.

5.52 (Computer-Assisted Instruction: Monitoring Student Performance) More sophisticated computer-assisted instruction systems monitor the student's performance over a period of time. The decision to begin a new topic is often based on the student's success with previous topics. Modify the program of Exercise 5.51 to count the number of correct and incorrect responses typed by the student. After the student types 10 answers, your program should calculate the percentage that are correct. If the percentage is lower than 75%, display "Please ask your teacher for extra help.", then reset the program so another student can try it. If the percentage is 75% or higher, display "Congratulations, you are ready to go to the next level!", then reset the program so another student can try it.

5.53 (Computer-Assisted Instruction: Difficulty Levels) Exercise 5.50 through Exercise 5.52 developed a computer-assisted instruction program to help teach an elementary school student multiplication. Modify the program to allow the user to enter a difficulty level. At a difficulty level of 1, the program should use only single-digit numbers in the problems; at a difficulty level of 2, numbers as large as two digits, and so on.

5.54 (Computer-Assisted Instruction: Varying the Types of Problems) Modify the program of Exercise 5.53 to allow the user to pick a type of arithmetic problem to study. An option of 1 means addition problems only, 2 means subtraction problems only, 3 means multiplication problems only and 4 means a random mixture of all these types.
C Arrays

Now go, write it before them in a table, and note it in a book.
—Isaiah 30:8

To go beyond is as wrong as to fall short.
—Confucius

Begin at the beginning, … and go on till you come to the end: then stop.
—Lewis Carroll

Objectives

In this chapter, you’ll learn:

- To use the array data structure to represent lists and tables of values.
- To define an array, initialize an array and refer to individual elements of an array.
- To define symbolic constants.
- To pass arrays to functions.
- To use arrays to store, sort and search lists and tables of values.
- To define and manipulate multiple-subscripted arrays.
6.1 Introduction

This chapter serves as an introduction to the important topic of data structures. Arrays are data structures consisting of related data items of the same type. In Chapter 10, we discuss C’s notion of struct (structure)—a data structure consisting of related data items of possibly different types. Arrays and structures are “static” entities in that they remain the same size throughout program execution (they may, of course, be of automatic storage class and hence created and destroyed each time the blocks in which they are defined are entered and exited). In Chapter 12, we introduce dynamic data structures such as lists, queues, stacks and trees that may grow and shrink as programs execute.

6.2 Arrays

An array is a group of memory locations related by the fact that they all have the same name and the same type. To refer to a particular location or element in the array, we specify the name of the array and the position number of the particular element in the array.

Figure 6.1 shows an integer array called c. This array contains 12 elements. Any one of these elements may be referred to by giving the name of the array followed by the position number of the particular element in square brackets ([]). The first element in every array is the zeroth element. Thus, the first element of array c is referred to as c[0], the second element of array c is referred to as c[1], the seventh element of array c is referred to as c[6], and, in general, the ith element of array c is referred to as c[i - 1]. Array names, like other variable names, can contain only letters, digits and underscores. Array names cannot begin with a digit.

The position number contained within square brackets is more formally called a subscript (or index). A subscript must be an integer or an integer expression. If a program uses an expression as a subscript, then the expression is evaluated to determine the subscript. For example, if a = 5 and b = 6, then the statement

\[
c[ a + b ] += 2;
\]

adds 2 to array element c[11]. A subscripted array name is an lvalue—it can be used on the left side of an assignment.

Let’s examine array c (Fig. 6.1) more closely. The array’s name is c. Its 12 elements are referred to as c[0], c[1], c[2], ..., c[11]. The value stored in c[0] is −45, the value of c[1] is 6, the value of c[2] is 0, the value of c[7] is 62 and the value of c[11] is 78. To print the sum of the values contained in the first three elements of array c, we’d write

\[
\text{printf( "\%d", c[0] + c[1] + c[2]);}
\]
To divide the value of the seventh element of array \( c \) by 2 and assign the result to the variable \( x \), we’d write

\[
x = c[6] / 2;
\]

**Common Programming Error 6.1**

It’s important to note the difference between the “seventh element of the array” and “array element seven.” Because array subscripts begin at 0, the “seventh element of the array” has a subscript of 6, while “array element seven” has a subscript of 7 and is actually the eighth element of the array. This is a source of “off-by-one” errors.

The brackets used to enclose the subscript of an array are actually considered to be an operator in C. They have the same level of precedence as the function call operator (i.e., the parentheses that are placed following a function name to call that function). Figure 6.2 shows the precedence and associativity of the operators introduced to this point in the text.
6.3 Defining Arrays

Arrays occupy space in memory. You specify the type of each element and the number of elements required by each array so that the computer may reserve the appropriate amount of memory. To tell the computer to reserve 12 elements for integer array \textit{c}, the definition

\begin{verbatim}
int c[12];
\end{verbatim}

is used. The following definition

\begin{verbatim}
int b[100], x[27];
\end{verbatim}

reserves 100 elements for integer array \textit{b} and 27 elements for integer array \textit{x}.

Arrays may contain other data types. For example, an array of type \texttt{char} can be used to store a character string. Character strings and their similarity to arrays are discussed in Chapter 8. The relationship between pointers and arrays is discussed in Chapter 7.

6.4 Array Examples

This section presents several examples that demonstrate how to define arrays, how to initialize arrays and how to perform many common array manipulations.

\textit{Defining an Array and Using a Loop to Initialize the Array’s Elements}

Figure 6.3 uses \texttt{for} statements to initialize the elements of a 10-element integer array \textit{n} to zeros and print the array in tabular format. The first \texttt{printf} statement (line 16) displays the column heads for the two columns printed in the subsequent \texttt{for} statement.

\begin{verbatim}
/* Fig. 6.3: fig06_03.c */
#include <stdio.h>
/* function main begins program execution */
int main( void )
{
    int c[12];

    /* Fig. 6.3 | Initializing an array */
    
    int b[100], x[27];

    /* Fig. 6.3 | Initializing the elements of an array to zeros. (Part 1 of 2.) */
    for (int j = 0; j < 10; j++)
        n[j] = 0;

    /* Fig. 6.3 | Initializing the elements of an array to zeros. (Part 2 of 2.) */
    for (int j = 0; j < 10; j++)
    {
        printf("%d\t", n[j]);
    }
}
\end{verbatim}
Initializing an Array in a Definition with an Initializer List

The elements of an array can also be initialized when the array is defined by following the
definition with an equals sign and braces, {}, containing a comma-separated list of initial-
izers. Figure 6.4 initializes an integer array with 10 values (line 9) and prints the array in
tabular format.

```c
int n[10]; /* n is an array of 10 integers */
int i; /* counter */
/* initialize elements of array n to 0 */
for ( i = 0; i < 10; i++ ) {
    n[i] = 0; /* set element at location i to 0 */
} /* end for */
/* output contents of array n in tabular format */
for ( i = 0; i < 10; i++ ) {
    printf( "%7d%13d
", i, n[i] );
} /* end for */
return 0; /* indicates successful termination */
/* use initializer list to initialize array n */
```
If there are fewer initializers than elements in the array, the remaining elements are initialized to zero. For example, the elements of the array \( n \) in Fig. 6.3 could have been initialized to zero as follows:

```c
int n[10] = {0};
```

This explicitly initializes the first element to zero and initializes the remaining nine elements to zero because there are fewer initializers than there are elements in the array. It’s important to remember that arrays are not automatically initialized to zero. You must at least initialize the first element to zero for the remaining elements to be automatically zeroed. This method of initializing the array elements to 0 is performed at compile time for static arrays and at runtime for automatic arrays.

**Common Programming Error 6.2**

>> Forgetting to initialize the elements of an array whose elements should be initialized.

The array definition

```c
int n[5] = {32, 27, 64, 18, 95, 14};
```

causes a syntax error because there are six initializers and only five array elements.

**Common Programming Error 6.3**

>> Providing more initializers in an array initializer list than there are elements in the array is a syntax error.

If the array size is omitted from a definition with an initializer list, the number of elements in the array will be the number of elements in the initializer list. For example,

```c
int n[] = {1, 2, 3, 4, 5};
```

would create a five-element array.
Specifying an Array’s Size with a Symbolic Constant and Initializing Array Elements with Calculations

Figure 6.5 initializes the elements of a 10-element array s to the values 2, 4, 6, …, 20 and prints the array in tabular format. The values are generated by multiplying the loop counter by 2 and adding 2.

```c
/* Fig. 6.5: fig06_05.c */
#include <stdio.h>
#define SIZE 10 /* maximum size of array */

int main( void )
{
    /* symbolic constant SIZE can be used to specify array size */
    int s[ SIZE ]; /* array s has SIZE elements */
    int j; /* counter */

    for ( j = 0; j < SIZE; j++ ) { /* set the values */
        s[ j ] = 2 + 2 * j;
    } /* end for */

    printf( "%s%13s
", "Element", "Value" );

    /* output contents of array s in tabular format */
    for ( j = 0; j < SIZE; j++ ) {
        printf( "%7d%13d\n", j, s[ j ] );
    } /* end for */

    return 0; /* indicates successful termination */
} /* end main */
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
</tr>
</tbody>
</table>

Fig. 6.5 | Initialize the elements of array s to the even integers from 2 to 20.

The `#define` preprocessor directive is introduced in this program. Line 4

```c
#define SIZE 10
```

defines a **symbolic constant** `SIZE` whose value is 10. A symbolic constant is an identifier that is replaced with **replacement text** by the C preprocessor before the program is compiled. When the program is preprocessed, all occurrences of the symbolic constant `SIZE`
are replaced with the replacement text 10. Using symbolic constants to specify array sizes makes programs more scalable. In Fig. 6.5, we could have the first for loop (line 13) fill a 1000-element array by simply changing the value of SIZE in the #define directive from 10 to 1000. If the symbolic constant SIZE had not been used, we’d have to change the program in three separate places to scale the program to handle 1000 array elements. As programs get larger, this technique becomes more useful for writing clear programs.

Common Programming Error 6.4
Ending a #define or #include preprocessor directive with a semicolon. Remember that preprocessor directives are not C statements.

If the #define preprocessor directive in line 4 is terminated with a semicolon, all occurrences of the symbolic constant SIZE in the program are replaced with the text 10; by the preprocessor. This may lead to syntax errors at compile time, or logic errors at execution time. Remember that the preprocessor is not C—it’s only a text manipulator.

Software Engineering Observation 6.1
Defining the size of each array as a symbolic constant makes programs more scalable.

Good Programming Practice 6.1
Use only uppercase letters for symbolic constant names. This makes these constants stand out in a program and reminds you that symbolic constants are not variables.

Good Programming Practice 6.2
In multiword symbolic constant names, use underscores to separate the words for readability.

Summing the Elements of an Array
Figure 6.6 sums the values contained in the 12-element integer array a. The for statement’s body (line 16) does the totaling.

```c
/* Fig. 6.6: fig06_06.c
   Compute the sum of the elements of the array */
#include <stdio.h>
#define SIZE 12

/* function main begins program execution */
int main( void )
{
    /* use initializer list to initialize array */
    int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
}
```

Fig. 6.6 | Computing the sum of the elements of an array. (Part 1 of 2.)
Using Arrays to Summarize Survey Results

Our next example uses arrays to summarize the results of data collected in a survey. Consider the problem statement.

Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10 (1 means awful and 10 means excellent). Place the 40 responses in an integer array and summarize the results of the poll.

This is a typical array application (see Fig. 6.7). We wish to summarize the number of responses of each type (i.e., 1 through 10). The array responses (line 17) is a 40-element array of the students' responses. We use an 11-element array frequency (line 14) to count the number of occurrences of each response. We ignore frequency[0] because it’s logical to have response 1 increment frequency[1] rather than frequency[0]. This allows us to use each response directly as the subscript in the frequency array.

```c
/* sum contents of array a */
for ( i = 0; i < SIZE; i++ ) {
    total += a[ i ];
} /* end for */
```

```
Total of array element values is 383
```

**Fig. 6.6** Computing the sum of the elements of an array. (Part 2 of 2.)

```c
int answer; /* counter to loop through 40 responses */
int rating; /* counter to loop through frequencies 1-10 */

int initialize frequency counters to 0 */
int frequency[ FREQUENCY_SIZE ] = { 0 };

/* place the survey responses in the responses array */
int responses[ RESPONSE_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
```
Chapter 6  C Arrays

The for loop (line 24) takes the responses one at a time from the array responses and use that value as subscript in array frequency to determine element to increment */

for ( answer = 0; answer < RESPONSE_SIZE; answer++ ) {
    ++frequency[ responses[ answer ] ];
} /* end for */

/* display results */
printf( "%s%17s\n", "Rating", "Frequency" );

/* output the frequencies in a tabular format */
for ( rating = 1; rating < FREQUENCY_SIZE; rating++ ) {
    printf( "%6d%17d\n", rating, frequency[ rating ] );
} /* end for */

return 0; /* indicates successful termination */
} /* end main */

Table: Frequency Count

<table>
<thead>
<tr>
<th>Rating</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

Fig. 6.7  | Student poll analysis program. (Part 2 of 2.)

Good Programming Practice 6.3
Strive for program clarity. Sometimes it may be worthwhile to trade off the most efficient use of memory or processor time in favor of writing clearer programs.

Performance Tip 6.1
Sometimes performance considerations far outweigh clarity considerations.

The for loop (line 24) takes the responses one at a time from the array responses and increments one of the 10 counters (frequency[1] to frequency[10]) in the frequency array. The key statement in the loop is line 25

++frequency[ responses[ answer ] ];

which increments the appropriate frequency counter depending on the value of responses[answer]. When the counter variable answer is 0, responses[answer] is 1, so ++frequency[ responses[answer]] is interpreted as

++frequency[ 1 ];
which increments array element one. When `answer` is 1, `responses[answer]` is 2, so
```c
++frequency[responses[answer]]; // is interpreted as
```
which increments array element two. When `answer` is 2, `responses[answer]` is 6, so
```c
++frequency[responses[answer]]; // is actually interpreted as
```
which increments array element six, and so on. Regardless of the number of responses processed in the survey, only an 11-element array is required (ignoring element zero) to summarize the results. If the data contained invalid values such as 13, the program would attempt to add 1 to `frequency[13]`. This would be outside the bounds of the array. *C has no array bounds checking to prevent the program from referring to an element that does not exist.* Thus, an executing program can “walk off” the end of an array without warning. You should ensure that all array references remain within the bounds of the array.

### Common Programming Error 6.6
Refriring to an element outside the array bounds.

### Error-Prevention Tip 6.1
When looping through an array, the array subscript should never go below 0 and should always be less than the total number of elements in the array (`size – 1`). Make sure the loop-terminating condition prevents accessing elements outside this range.

### Error-Prevention Tip 6.2
Programs should validate the correctness of all input values to prevent erroneous information from affecting a program’s calculations.

### Graphing Array Element Values with Histograms
Our next example (Fig. 6.8) reads numbers from an array and graphs the information in the form of a bar chart or histogram—each number is printed, then a bar consisting of that many asterisks is printed beside the number. The nested `for` statement (line 20) draws the bars. Note the use of `printf("\n") to end a histogram bar (line 24).

```c
/* Fig. 6.8: fig06_08.c
Histogram printing program */
#include <stdio.h>
#define SIZE 10

/* function main begins program execution */
int main( void )
{
    /* use initializer list to initialize array n */
    int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
    int i; /* outer for counter for array elements */
    int j; /* inner for counter counts *s in each histogram bar */
```
Rolling a Die 6000 Times and Summarizing the Results in an Array

In Chapter 5, we stated that we’d show a more elegant method of writing the dice-rolling program of Fig. 5.8. The problem was to roll a single six-sided die 6000 times to test whether the random number generator actually produces random numbers. An array version of this program is shown in Fig. 6.9.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 7  

int main(void)  
{  
    int face; /* random die value 1 - 6 */  
    int roll; /* roll counter 1-6000 */  
    int frequency[SIZE] = { 0 }; /* clear counts */
```

---

**Fig. 6.8**  Histogram printing. (Part 2 of 2.)

**Fig. 6.9**  Dice-rolling program using an array instead of switch. (Part 1 of 2.)
Using Character Arrays to Store and Manipulate Strings

We have discussed only integer arrays. However, arrays are capable of holding data of any type. We now discuss storing strings in character arrays. So far, the only string-processing capability we have is outputting a string with `printf`. A string such as "hello" is really a static array of individual characters in C.

Character arrays have several unique features. A character array can be initialized using a string literal. For example,

```c
char string1[] = "first";
```
initializes the elements of array `string1` to the individual characters in the string literal "first". In this case, the size of array `string1` is determined by the compiler based on the length of the string. The string "first" contains five characters plus a special string-termination character called the null character. Thus, array `string1` actually contains six elements. The character constant representing the null character is '\0'. All strings in C end with this character. A character array representing a string should always be defined large enough to hold the number of characters in the string and the terminating null character.

Character arrays also can be initialized with individual character constants in an initializer list. The preceding definition is equivalent to

```c
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

Because a string is really an array of characters, we can access individual characters in a string directly using array subscript notation. For example, `string1[0]` is the character 'f' and `string1[3]` is the character 's'.

---

Fig. 6.9 | Dice-rolling program using an array instead of switch. (Part 2 of 2.)
We also can input a string directly into a character array from the keyboard using `scanf` and the conversion specifier %s. For example,

```
char string2[ 20 ];
```

creates a character array capable of storing a string of at most 19 characters and a terminating null character. The statement

```
scanf( "%s", string2 );
```

reads a string from the keyboard into `string2`. The name of the array is passed to `scanf` without the preceding & used with nonstring variables. The & is normally used to provide `scanf` with a variable’s location in memory so that a value can be stored there. In Section 6.5, when we discuss passing arrays to functions, we’ll see that the value of an array name is the address of the start of the array; therefore, the & is not necessary. Function `scanf` will read characters until a space, tab, newline or end-of-file indicator is encountered. The string should be no longer than 19 characters to leave room for the terminating null character. If the user types 20 or more characters, your program may crash! For this reason, use the conversion specifier %19s so that `scanf` does not write characters into memory beyond the end of the array.

It’s your responsibility to ensure that the array into which the string is read is capable of holding any string that the user types at the keyboard. Function `scanf` reads characters from the keyboard until the first white-space character is encountered—it does not check how large the array is. Thus, `scanf` can write beyond the end of the array.

**Common Programming Error 6.7**

Not providing `scanf` with a character array large enough to store a string typed at the keyboard can result in destruction of data in a program and other runtime errors. This can also make a system susceptible to worm and virus attacks.

A character array representing a string can be output with `printf` and the %s conversion specifier. The array `string2` is printed with the statement

```
printf( "%s\n", string2 );
```

Function `printf`, like `scanf`, does not check how large the character array is. The characters of the string are printed until a terminating null character is encountered.

Figure 6.10 demonstrates initializing a character array with a string literal, reading a string into a character array, printing a character array as a string and accessing individual characters of a string.

```
/* Fig. 6.10: fig06_10.c */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    char string1[ 20 ]; /* reserves 20 characters */
    char string2[] = "string literal"; /* reserves 15 characters */
```

**Fig. 6.10** | Treating character arrays as strings. (Part 1 of 2.)
6.4 Array Examples

Figure 6.10 uses a `for` statement (line 22) to loop through the `string1` array and print the individual characters separated by spaces, using the `%c` conversion specifier. The condition in the `for` statement, `string1[i] != '\0'`, is true while the terminating null character has not been encountered in the string.

Static Local Arrays and Automatic Local Arrays
Chapter 5 discussed the storage-class specifier `static`. A `static` local variable exists for the duration of the program, but is visible only in the function body. We can apply `static` to a local array definition so the array is not created and initialized each time the function is called and the array is not destroyed each time the function is exited in the program. This reduces program execution time, particularly for programs with frequently called functions that contain large arrays.

**Performance Tip 6.2**
In functions that contain automatic arrays where the function is in and out of scope frequently, make the array `static` so it’s not created each time the function is called.

Arrays that are `static` are initialized once at compile time. If you do not explicitly initialize a static array, that array’s elements are initialized to zero by the compiler.

Figure 6.11 demonstrates function `staticArrayInit` (line 22) with a local `static` array (line 25) and function `automaticArrayInit` (line 44) with a local automatic array
Function staticArrayInit is called twice (lines 12 and 16). The local static array in the function is initialized to zero by the compiler (line 25). The function prints the array, adds 5 to each element and prints the array again. The second time the function is called, the static array contains the values stored during the first function call. Function automaticArrayInit is also called twice (lines 13 and 17). The elements of the automatic local array in the function are initialized with the values 1, 2 and 3 (line 47). The function prints the array, adds 5 to each element and prints the array again. The second time the function is called, the array elements are initialized to 1, 2 and 3 again because the array has automatic storage duration.

Common Programming Error 6.8
Assuming that elements of a local static array are initialized to zero every time the function in which the array is defined is called.

```c
/* Fig. 6.11: fig06_11.c
Static arrays are initialized to zero */
#include <stdio.h>

void staticArrayInit( void ); /* function prototype */
void automaticArrayInit( void ); /* function prototype */

/* function main begins program execution */
int main( void )
{
    printf( "First call to each function:\n" );
    staticArrayInit();
    automaticArrayInit();

    printf( "Second call to each function:\n" );
    staticArrayInit();
    automaticArrayInit();
    return 0; /* indicates successful termination */
} /* end main */

/* function to demonstrate a static local array */
void staticArrayInit( void )
{
    /* initializes elements to 0 first time function is called */
    static int array1[3];
    int i; /* counter */

    printf( "Values on entering staticArrayInit:\n" );

    /* output contents of array1 */
    for ( i = 0; i <= 2; i++ )
        printf( "array1[ %d ] = %d \n", i, array1[ i ] );
    /* end for */

    printf( "Values on exiting staticArrayInit:\n" );
}
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 1 of 2.)
/* modify and output contents of array1 */
for ( i = 0; i <= 2; i++ ) {
    printf( "array1[ %d ] = %d  ", i, array1[ i ] += 5 );
} /* end function staticArrayInit */

/* function to demonstrate an automatic local array */
void automaticArrayInit( void )
{
    /* initializes elements each time function is called */
    int i; /* counter */
    printf( "

Values on entering automaticArrayInit:
" );
    for ( i = 0; i <= 2; i++ ) {
        printf("array2[ %d ] = %d  ", i, array2[ i ]);
    } /* end for */
    printf( "
Values on exiting automaticArrayInit:
" );
    for ( i = 0; i <= 2; i++ ) {
        printf("array2[ %d ] = %d  ", i, array2[ i ] += 5 );
    } /* end for */
} /* end function automaticArrayInit */

First call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 0  array1[ 1 ] = 0  array1[ 2 ] = 0
Values on exiting staticArrayInit:

Values on entering automaticArrayInit:
Values on exiting automaticArrayInit:

Second call to each function:

Values on entering staticArrayInit:
Values on exiting staticArrayInit:

Values on entering automaticArrayInit:
Values on exiting automaticArrayInit:

Fig. 6.11  |  Static arrays are initialized to zero if not explicitly initialized. (Part 2 of 2.)
6.5 Passing Arrays to Functions

To pass an array argument to a function, specify the name of the array without any brackets. For example, if array `hourlyTemperatures` has been defined as

```c
int hourlyTemperatures[24];
```

the function call

```c
modifyArray(hourlyTemperatures, 24)
```

passes array `hourlyTemperatures` and its size to function `modifyArray`. Unlike char arrays that contain strings, other array types do not have a special terminator. For this reason, the size of an array is passed to the function, so that the function can process the proper number of elements.

C automatically passes arrays to functions by reference—the called functions can modify the element values in the callers’ original arrays. The name of the array evaluates to the address of the first element of the array. Because the starting address of the array is passed, the called function knows precisely where the array is stored. Therefore, when the called function modifies array elements in its function body, it’s modifying the actual elements of the array in their original memory locations.

Figure 6.12 demonstrates that an array name is really the address of the first element of an array by printing array, `&array[0]` and `&array` using the `%p` conversion specifier—a special conversion specifier for printing addresses. The `%p` conversion specifier normally outputs addresses as hexadecimal numbers. Hexadecimal (base 16) numbers consist of the digits 0 through 9 and the letters A through F (these letters are the hexadecimal equivalents of the numbers 10–15). They are often used as shorthand notation for large integer values. Appendix C, Number Systems, provides an in-depth discussion of the relationships among binary (base 2), octal (base 8), decimal (base 10; standard integers) and hexadecimal integers. The output shows that both `array` and `&array[0]` have the same value, namely `0012FF78`. The output of this program is system dependent, but the addresses are always identical for a particular execution of this program on a particular computer.

**Performance Tip 6.3**

Passing arrays by reference makes sense for performance reasons. If arrays were passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would be time consuming and would consume storage for the copies of the arrays.

**Software Engineering Observation 6.2**

It’s possible to pass an array by value (by using a simple trick we explain in Chapter 10).

Although entire arrays are passed by reference, individual array elements are passed by value exactly as simple variables are. Such simple single pieces of data (such as individual ints, floats and chars) are called scalars. To pass an element of an array to a function, use the subscripted name of the array element as an argument in the function call. In Chapter 7, we show how to pass scalars (i.e., individual variables and array elements) to functions by reference.
For a function to receive an array through a function call, the function’s parameter list must specify that an array will be received. For example, the function header for function modifyArray (that we called earlier in this section) might be written as indicating that modifyArray expects to receive an array of integers in parameter b and the number of array elements in parameter size. The size of the array is not required between the array brackets. If it’s included, the compiler checks that it’s greater than zero, then ignores it. Specifying a negative size is a compilation error. Because arrays are automatically passed by reference, when the called function uses the array name b, it will be referring to the array in the caller (array hourlyTemperatures in the preceding call). In Chapter 7, we introduce other notations for indicating that an array is being received by a function. As we’ll see, these notations are based on the intimate relationship between arrays and pointers in C.

Figure 6.13 demonstrates the difference between passing an entire array and passing an array element. The program first prints the five elements of integer array a (lines 20–22). Next, a and its size are passed to function modifyArray (line 27), where each of a’s elements is multiplied by 2 (lines 54–55). Then a is reprinted in main (lines 32–34). As the output shows, the elements of a are indeed modified by modifyArray. Now the program prints the value of a[3] (line 38) and passes it to function modifyElement (line 40). Function modifyElement multiplies its argument by 2 (line 64) and prints the new value. When a[3] is reprinted in main (line 43), it has not been modified, because individual array elements are passed by value.

```c
/* Fig. 6.12: fig06_12.c */
#include <stdio.h>

int main( void )
{
    char array[5]; /* define an array of size 5 */
    printf("    array = %p
&array[0] = %p
   &array = %p
", array, &array[0], &array);
    return 0; /* indicates successful termination */
} /* end main */
```

Fig. 6.12 | Array name is the same as the address of the array’s first element.

```c
/* Fig. 6.13: fig06_13.c */
#include <stdio.h>
#define SIZE 5

void modifyArray( int b[], int size )
```

```c
/* Fig. 6.13: fig06_13.c */
#include <stdio.h>
#define SIZE 5

/* Passing arrays and individual array elements to functions */
#include <stdio.h>

/* Passing arrays and individual array elements to functions */
#include <stdio.h>
define SIZE 5
```
/* function prototypes */
void modifyArray( int b[], int size );
void modifyElement( int e );

/* function main begins program execution */
int main( void )
{
    int a[ SIZE ] = { 0, 1, 2, 3, 4 }; /* initialize a */
    int i; /* counter */

    printf( "Effects of passing entire array by reference:
            The values of the original array are:
            ");

    /* output original array */
    for ( i = 0; i < SIZE; i++ ) {
        printf( "%3d", a[ i ] );
    } /* end for */

    printf( "\n\n" );

    /* pass array a to modifyArray by reference */
    modifyArray( a, SIZE );

    printf( "The values of the modified array are:\n" );

    /* output modified array */
    for ( i = 0; i < SIZE; i++ ) {
        printf( "%3d", a[ i ] );
    } /* end for */

    /* output value of a[ 3 ] */
    printf( "\n\nEffects of passing array element "
            "by value:\n            The value of a[3] is %d", a[ 3 ] );

    modifyElement( a[ 3 ] ); /* pass array element a[ 3 ] by value */

    /* output value of a[ 3 ] */
    printf( "The value of a[ 3 ] is %d", a[ 3 ] );
    return 0; /* indicates successful termination */
} /* end main */

/* in function modifyArray, "b" points to the original array "a" in memory */
void modifyArray( int b[], int size )
{
    int j; /* counter */

    /* multiply each array element by 2 */
    for ( j = 0; j < size; j++ ) {
        b[ j ] *= 2;
    } /* end for */
} /* end function modifyArray */

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 2 of 3.)
There may be situations in your programs in which a function should not be allowed to modify array elements. Because arrays are always passed by reference, modification of values in an array is difficult to control. C provides the type qualifier `const` to prevent modification of array values in a function. When an array parameter is preceded by the `const` qualifier, the array elements become constant in the function body, and any attempt to modify an element of the array in the function body results in a compile-time error. This enables you to correct a program so it does not attempt to modify array elements.

Figure 6.14 demonstrates the `const` type qualifier. Function `tryToModifyArray` (line 20) is defined with parameter `const int b[]`, which specifies that array `b` is constant and cannot be modified. The output shows the error messages produced by the compiler—the errors may be different on your system. Each of the three attempts by the function to modify array elements results in the compiler error “l-value specifies a const object.” The `const` qualifier is discussed again in Chapter 7.

Effects of passing entire array by reference:

The values of the original array are:

\[0 \; 1 \; 2 \; 3 \; 4\]

The values of the modified array are:

\[0 \; 2 \; 4 \; 6 \; 8\]

Effects of passing array element by value:

The value of a[3] is 6

Value in modifyElement is 12

The value of a[ 3 ] is 6

---

```c
/* Fig. 6.14: fig06_14.c
Demonstrating the const type qualifier with arrays */
#include <stdio.h>
void tryToModifyArray( const int b[] ); /* function prototype */
/* function main begins program execution */
int main( void )
{
    int a[] = { 10, 20, 30 }; /* initialize a */

    /* in function modifyElement, "e" is a local copy of array element a[ 3 ] passed from main */
    void modifyElement( int e )
    {
        /* multiply parameter by 2 */
        printf( "Value in modifyElement is %d\n", e *= 2 );
    } /* end function modifyElement */
```

---

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 3 of 3.)
Chapter 6  C Arrays

6.6 Sorting Arrays

Sorting data (i.e., placing the data into a particular order such as ascending or descending) is one of the most important computing applications. A bank sorts all checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, within that, by first name to make it easy to find phone numbers. Virtually every organization must sort some data and in many cases massive amounts of data. Sorting data is an intriguing problem which has attracted some of the most intense research efforts in the field of computer science. In this chapter we discuss what is perhaps the simplest known sorting scheme. In Chapter 12 and Appendix F, we investigate more complex schemes that yield superior performance.

Figure 6.15 sorts the values in the elements of the 10-element array a (line 10) into ascending order. The technique we use is called the bubble sort or the sinking sort because the smaller values gradually “bubble” their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array. The technique

```c
tryToModifyArray( a );
printf("%d %d %d\n", a[0], a[1], a[2]);
return 0; /* indicates successful termination */
}
/* end main */

// in function tryToModifyArray, array b is const, so it cannot be used to modify the original array a in main. */

void tryToModifyArray( const int b[] )
{
    b[0] /= 2; /* error */
    b[1] /= 2; /* error */
    b[2] /= 2; /* error */
}
/* end function tryToModifyArray */
```

Fig. 6.14 | const type qualifier. (Part 2 of 2.)

**Software Engineering Observation 6.3**

The const type qualifier can be applied to an array parameter in a function definition to prevent the original array from being modified in the function body. This is another example of the principle of least privilege. Functions should not be given the capability to modify an array unless it’s absolutely necessary.

**6.6 Sorting Arrays**

Sorting data (i.e., placing the data into a particular order such as ascending or descending) is one of the most important computing applications. A bank sorts all checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, within that, by first name to make it easy to find phone numbers. Virtually every organization must sort some data and in many cases massive amounts of data. Sorting data is an intriguing problem which has attracted some of the most intense research efforts in the field of computer science. In this chapter we discuss what is perhaps the simplest known sorting scheme. In Chapter 12 and Appendix F, we investigate more complex schemes that yield superior performance.

**Performance Tip 6.4**

Often, the simplest algorithms perform poorly. Their virtue is that they are easy to write, test and debug. More complex algorithms are often needed to realize maximum performance.

Figure 6.15 sorts the values in the elements of the 10-element array a (line 10) into ascending order. The technique we use is called the bubble sort or the sinking sort because the smaller values gradually “bubble” their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array. The technique
is to make several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or if the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.

```c
/* Fig. 6.15: fig06_15.c */
/* This program sorts an array's values into ascending order */
#include <stdio.h>
#define SIZE 10

/* function main begins program execution */
int main( void )
{
    /* initialize a */
    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
    int pass; /* passes counter */
    int i; /* comparisons counter */
    int hold; /* temporary location used to swap array elements */

    printf( "Data items in original order\n" );
    /* output original array */
    for ( i = 0; i < SIZE; i++ ) {
        printf( "%4d", a[ i ] );
    } /* end for */

    /* bubble sort */
    /* loop to control number of passes */
    for ( pass = 1; pass < SIZE; pass++ ) {
        /* loop to control number of comparisons per pass */
        for ( i = 0; i < SIZE - 1; i++ ) {
            /* compare adjacent elements and swap them if first
               element is greater than second element */
            if ( a[ i ] > a[ i + 1 ] ) {
                hold = a[ i ];
                a[ i ] = a[ i + 1 ];
                a[ i + 1 ] = hold;
            } /* end if */
        } /* end inner for */
    } /* end outer for */

    printf( "\nData items in ascending order\n" );
    /* output sorted array */
    for ( i = 0; i < SIZE; i++ ) {
        printf( "%4d", a[ i ] );
    } /* end for */

    printf( "\n" );
    return 0; /* indicates successful termination */
} /* end main */
```

Fig. 6.15  Sorting an array with bubble sort. (Part 1 of 2.)
First the program compares \( a[0] \) to \( a[1] \), then \( a[1] \) to \( a[2] \), then \( a[2] \) to \( a[3] \), and so on until it completes the pass by comparing \( a[8] \) to \( a[9] \). Although there are 10 elements, only nine comparisons are performed. Because of the way the successive comparisons are made, a large value may move down the array many positions on a single pass, but a small value may move up only one position. On the first pass, the largest value is guaranteed to sink to the bottom element of the array, \( a[9] \). On the second pass, the second-largest value is guaranteed to sink to \( a[8] \). On the ninth pass, the ninth-largest value sinks to \( a[1] \). This leaves the smallest value in \( a[0] \), so only nine passes of the array are needed to sort the array, even though there are ten elements.

The sorting is performed by the nested for loop (lines 24–37). If a swap is necessary, it’s performed by the three assignments

\[
\text{hold} = a[i]; \\
a[i] = a[i+1]; \\
a[i+1] = \text{hold};
\]

where the extra variable \( \text{hold} \) temporarily stores one of the two values being swapped. The swap cannot be performed with only the two assignments

\[
\text{hold} = a[i]; \\
a[i] = a[i+1]; \\
a[i+1] = a[i];
\]

If, for example, \( a[i] \) is 7 and \( a[i+1] \) is 5, after the first assignment both values will be 5 and the value 7 will be lost. Hence the need for the extra variable \( \text{hold} \).

The chief virtue of the bubble sort is that it’s easy to program. However, the bubble sort runs slowly because every exchange moves an element only one position closer to its final destination. This becomes apparent when sorting large arrays. In the exercises, we’ll develop more efficient versions of the bubble sort. Far more efficient sorts than the bubble sort have been developed. We’ll investigate a few of these in the exercises. More advanced courses investigate sorting and searching in greater depth.

### 6.7 Case Study: Computing Mean, Median and Mode Using Arrays

We now consider a larger example. Computers are commonly used for survey data analysis to compile and analyze the results of surveys and opinion polls. Figure 6.16 uses array response initialized with 99 responses to a survey. Each response is a number from 1 to 9. The program computes the mean, median and mode of the 99 values.

The mean is the arithmetic average of the 99 values. Function mean (line 40) computes the mean by totaling the 99 elements and dividing the result by 99.

The median is the “middle value.” Function median (line 61) determines the median by calling function bubbleSort (defined in line 133) to sort the array of responses into
ascending order, then picking the middle element, answer[SIZE / 2], of the sorted array. When there is an even number of elements, the median should be calculated as the mean of the two middle elements. Function median does not currently provide this capability. Function printArray (line 156) is called to output the response array.

The mode is the value that occurs most frequently among the 99 responses. Function mode (line 82) determines the mode by counting the number of responses of each type, then selecting the value with the greatest count. This version of function mode does not handle a tie (see Exercise 6.14). Function mode also produces a histogram to aid in determining the mode graphically. Figure 6.17 contains a sample run of this program. This example includes most of the common manipulations usually required in array problems, including passing arrays to functions.

```c
/* Fig. 6.16: fig06_16.c
This program introduces the topic of survey data analysis.
It computes the mean, median and mode of the data */
#include <stdio.h>
#define SIZE 99

/* function prototypes */
void mean( const int answer[] );
void median( int answer[] );
void mode( int freq[], const int answer[] );
void bubbleSort( int a[] );
void printArray( const int a[] );

/* function main begins program execution */
int main( void )
{
    int frequency[10] = {0}; /* initialize array frequency */
    int response[SIZE] =
    { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
        7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
        6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
        7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
        6, 7, 8, 7, 8, 7, 9, 8, 2,
        7, 8, 9, 8, 9, 8, 9, 7, 3, 5,
        5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
        7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
        7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
        4, 5, 6, 1, 6, 5, 7, 8, 7};
    /* process responses */
    mean( response );
    median( response );
    mode( frequency, response );
    return 0; /* indicates successful termination */
} /* end main */
```

Fig. 6.16  |  Survey data analysis program. (Part 1 of 4.)
/* calculate average of all response values */
void mean( const int answer[] )
{
    int j; /* counter for totaling array elements */
    int total = 0; /* variable to hold sum of array elements */
    printf( "%s\n%s\n%s\n", "********", " Mean", "********" );
    /* total response values */
    for ( j = 0; j < SIZE; j++ ) {
        total += answer[ j ];
    } /* end for */
    printf( "The mean is the average value of the data\n" "items. The mean is equal to the total of\" "all the data items divided by the number\" "of data items ( %d ). The mean value for\" "this run is: %d / %d = %.4f\n\n", SIZE, total, SIZE, ( double ) total / SIZE );
} /* end function mean */

/* sort array and determine median element's value */
void median( int answer[] )
{
    printf( "%s\n%s\n%s\n", "********", " Median", "********", "The unsorted array of responses is" );
    printArray( answer ); /* output unsorted array */
    bubbleSort( answer ); /* sort array */
    printf( "$\n\nThe sorted array is$" );
    printArray( answer ); /* output sorted array */
    /* display median element */
    printf( "$\n\nThe median is element %d of\n" "$ the sorted %d element array.$", SIZE / 2, SIZE, answer[ SIZE / 2 ] );
} /* end function median */

/* determine most frequent response */
void mode( int freq[], const int answer[] )
{
    int rating; /* counter for accessing elements 1-9 of array freq */
    int j; /* counter for summarizing elements 0-98 of array answer */
    int h; /* counter for displaying histograms of elements in array freq */
    int largest = 0; /* represents largest frequency */
    int modeValue = 0; /* represents most frequent response */
    printf( "%s\n%s\n%s\n", "********", " Mode", "********" );

Fig. 6.16  |  Survey data analysis program. (Part 2 of 4.)
/* initialize frequencies to 0 */
for (rating = 1; rating <= 9; rating++) {
    freq[rating] = 0;
} /* end for */

/* summarize frequencies */
for (j = 0; j < SIZE; j++) {
    ++freq[answer[j]];
} /* end for */

/* output headers for result columns */
printf("%s%11s%19s\n\n%54s
%54s
\n\n", "Response", "Frequency", "Histogram", "1 1 2 2", "5 0 5 0 5");

/* output results */
for (rating = 1; rating <= 9; rating++) {
    printf("%8d%11d          ", rating, freq[rating]);
    for (h = 1; h <= freq[rating]; h++) {
        printf("*");
    } /* end inner for */
    printf("\n"); /* being new line of output */
} /* end outer for */

/* display the mode value */
printf("The mode is the most frequent value.\n" "For this run the mode is %d which occurred" " %d times.\n", modeValue, largest);

/* function that sorts an array with bubble sort algorithm */
void bubbleSort( int a[] )
{
    int pass; /* pass counter */
    int j; /* comparison counter */
    int hold; /* temporary location used to swap elements */
    /* loop to control number of passes */
    for (pass = 1; pass < SIZE; pass++ ) {
        /* loop to control number of comparisons per pass */
        for (j = 0; j < SIZE - 1; j++) {
            /* check all possible locations */
            if (a[j] > a[j + 1]) {
                hold = a[j]; /* temporary location used to swap elements */
                a[j] = a[j + 1]; /* temporary location used to swap elements */
                a[j + 1] = hold; /* temporary location used to swap elements */
            }
        } /* end inner for */
    } /* end outer for */
}

/* function to compute mean, median and mode */
float mean(float arr[], int n)
{
    float sum = 0.0;
    for (int i = 0; i < n; i++)
        sum += arr[i];
    return sum / n;
}

/* function to compute median */
float median(float arr[], int n)
{
    int mid = n / 2;
    if (n % 2) return arr[mid];
    else return (arr[mid - 1] + arr[mid]) / 2;
}

/* function to compute mode */
int mode(float arr[], int n)
{
    int max_count = 0, mode = arr[0];
    int count;
    for (int i = 0; i < n; i++)
    {
        count = 0;
        for (int j = 0; j < n; j++)
        {
            if (arr[i] == arr[j])
                count++;
        }
        if (count > max_count)
        {
            max_count = count;
            mode = arr[i];
        }
    }
    return mode;
}
/* swap elements if out of order */
if ( a[ j ] > a[ j + 1 ] ) {
    hold = a[ j ];
    a[ j ] = a[ j + 1 ];
    a[ j + 1 ] = hold;
} /* end if */
} /* end inner for */
} /* end outer for */
} /* end function bubbleSort */

void printArray( const int a[] )
{
    int j; /* counter */
    /* output array contents */
    for ( j = 0; j < SIZE; j++ ) {
        if ( j % 20 == 0 ) { /* begin new line every 20 values */
            printf("\n");
        } /* end if */
        printf("%2d", a[ j ] );
    } /* end for */
} /* end function printArray */

********
Mean
********
The mean is the average value of the data items. The mean is equal to the total of all the data items divided by the number of data items (99). The mean value for this run is: 681 / 99 = 6.8788

********
Median
********
The unsorted array of responses is
6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
6 7 8 9 3 9 8 7 8 7 8 9 8 9 8 9 7 8 9
6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
1 2 2 2 3 3 3 3 4 4 4 4 4 4 5 5 5 5 5 5 5
5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8

Fig. 6.16 | Survey data analysis program. (Part 4 of 4.)
You’ll often work with large amounts of data stored in arrays. It may be necessary to determine whether an array contains a value that matches a certain key value. The process of finding a particular element of an array is called searching. In this section we discuss two searching techniques—the simple linear search technique and the more efficient (but more complex) binary search technique. Exercise 6.32 and Exercise 6.33 at the end of this chapter ask you to implement recursive versions of the linear search and the binary search.

**Searching an Array with Linear Search**

The linear search (Fig. 6.18) compares each element of the array with the search key. Since the array is not in any particular order, it’s just as likely that the value will be found in the first element as in the last. On average, therefore, the program will have to compare the search key with half the elements of the array.

```c
/* Fig. 6.18: fig06_18.c
   Linear search of an array */
#include <stdio.h>
#define SIZE 100
```

**Fig. 6.18** | Linear search of an array. (Part 1 of 3.)
Chapter 6  C Arrays

/* function prototype */

int linearSearch( const int array[], int key, int size );

/* function main begins program execution */

int main( void )

{
    int a[ SIZE ]; /* create array a */
    int x; /* counter for initializing elements 0-99 of array a */
    int searchKey; /* value to locate in array a */
    int element; /* variable to hold location of searchKey or -1 */

    /* create data */
    for ( x = 0; x < SIZE; x++ ) {
        a[ x ] = 2 * x;
    } /* end for */

    printf( "Enter integer search key:\n" );
    scanf( "%d", &searchKey );

    /* attempt to locate searchKey in array a */
    element = linearSearch( a, searchKey, SIZE );

    /* display results */
    if ( element != -1 ) {
        printf( "Found value in element %d\n", element );
    } /* end if */
    else {
        printf( "Value not found\n" );
    } /* end else */

    return 0; /* indicates successful termination */
} /* end main */

/* compare key to every element of array until the location is found
or until the end of array is reached; return subscript of element
if key or -1 if key is not found */

int linearSearch( const int array[], int key, int size )

{
    int n; /* counter */

    /* loop through array */
    for ( n = 0; n < size; ++n ) {

        if ( array[ n ] == key ) {
            return n; /* return location of key */
        } /* end if */
    } /* end for */

    return -1; /* key not found */
} /* end function linearSearch */

Fig. 6.18  |  Linear search of an array. (Part 2 of 3.)
Searching Arrays

Searching an Array with Binary Search

The linear searching method works well for small or unsorted arrays. However, for large arrays linear searching is inefficient. If the array is sorted, the high-speed binary search technique can be used.

The binary search algorithm eliminates from consideration one-half of the elements in a sorted array after each comparison. The algorithm locates the middle element of the array and compares it to the search key. If they are equal, the search key is found and the array subscript of that element is returned. If they are not equal, the problem is reduced to searching one-half of the array. If the search key is less than the middle element of the array, the first half of the array is searched, otherwise the second half of the array is searched. If the search key is not found in the specified subarray (piece of the original array), the algorithm is repeated on one-quarter of the original array. The search continues until the search key is equal to the middle element of a subarray, or until the subarray consists of one element that is not equal to the search key (i.e., the search key is not found).

In a worst case-scenario, searching an array of 1023 elements takes only 10 comparisons using a binary search. Repeatedly dividing 1024 by 2 yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1. The number 1024 (210) is divided by 2 only 10 times to get the value 1. Dividing by 2 is equivalent to one comparison in the binary search algorithm. An array of 1048576 (220) elements takes a maximum of 20 comparisons to find the search key. An array of one billion elements takes a maximum of 30 comparisons to find the search key. This is a tremendous increase in performance over the linear search that required comparing the search key to an average of half of the array elements. For a one-billion-element array, this is a difference between an average of 500 million comparisons and a maximum of 30 comparisons! The maximum comparisons for any array can be determined by finding the first power of 2 greater than the number of array elements.

Figure 6.19 presents the iterative version of function binarySearch (lines 44–74). The function receives four arguments—an integer array b to be searched, an integer searchKey, the low array subscript and the high array subscript (these define the portion of the array to be searched). If the search key does not match the middle element of a subarray, the low subscript or high subscript is modified so that a smaller subarray can be searched. If the search key is less than the middle element, the high subscript is set to middle - 1 and the search is continued on the elements from low to middle - 1. If the search key is greater than the middle element, the low subscript is set to middle + 1 and the search is continued on the elements from middle + 1 to high. The program uses an array of 15 elements. The first power of 2 greater than the number of elements in this array

Enter integer search key: 36
Found value in element 18

Enter integer search key: 37
Value not found
is $16 (2^4)$, so a maximum of 4 comparisons are required to find the search key. The program uses function `printHeader` (lines 77–96) to output the array subscripts and function `printRow` (lines 100–120) to output each subarray during the binary search process. The middle element in each subarray is marked with an asterisk (*) to indicate the element to which the search key is compared.

```c
/* Fig. 6.19: fig06_19.c */
#include <stdio.h>
#define SIZE 15

/* function prototypes */
int binarySearch( const int b[], int searchKey, int low, int high );
void printHeader( void );
void printRow( const int b[], int low, int mid, int high );

/* function main begins program execution */
int main( void )
{
    int a[ SIZE ]; /* create array a */
    int i; /* counter for initializing elements 0-14 of array a */
    int key; /* value to locate in array a */
    int result; /* variable to hold location of key or -1 */

    /* create data */
    for ( i = 0; i < SIZE; i++ ) {
        a[ i ] = 2 * i;
    } /* end for */

    printf( "Enter a number between 0 and 28: ");
    scanf( "%d", &key );

    printHeader();

    /* search for key in array a */
    result = binarySearch( a, key, 0, SIZE - 1 );

    /* display results */
    if ( result != -1 ) {
        printf( "\n%d found in array element %d\n", key, result );
    } /* end if */
    else {
        printf( "\n%d not found\n", key );
    } /* end else */

    return 0; /* indicates successful termination */
} /* end main */

/* function to perform binary search of an array */
int binarySearch( const int b[], int searchKey, int low, int high )
{
```

Fig. 6.19  |  Binary search of a sorted array. (Part 1 of 4.)
int middle; /* variable to hold middle element of array */

/* loop until low subscript is greater than high subscript */
while ( low <= high ) {

    /* determine middle element of subarray being searched */
middle = ( low + high ) / 2;

    /* display subarray used in this loop iteration */
    printRow( b, low, middle, high );

    /* if searchKey matched middle element, return middle */
    if ( searchKey == b[ middle ] ) {
        return middle;
    } /* end if */

    /* if searchKey less than middle element, set new high */
    else if ( searchKey < b[ middle ] ) {
        high = middle - 1; /* search low end of array */
    } /* end else if */

    /* if searchKey greater than middle element, set new low */
    else {
        low = middle + 1; /* search high end of array */
    } /* end else */

    return -1; /* searchKey not found */
} /* end function binarySearch */

/* Print a header for the output */
void printHeader( void )
{
    int i; /* counter */
    printf( "\nSubscripts:\n" );

    /* output column head */
    for ( i = 0; i < SIZE; i++ ) {
        printf( "%d " , i );
    } /* end for */

    printf( "\n" ); /* start new line of output */

    /* output line of - characters */
    for ( i = 1; i <= 4 * SIZE; i++ ) {
        printf( "-" );
    } /* end for */

    printf( "\n" ); /* start new line of output */
} /* end function printHeader */
/* Print one row of output showing the current part of the array being processed. */

void printRow(const int b[], int low, int mid, int high)
{
    int i; /* counter for iterating through array b */
    /* loop through entire array */
    for (i = 0; i < SIZE; i++) {
        /* display spaces if outside current subarray range */
        if (i < low || i > high) {
            printf("    ");
        } /* end if */
        else if (i == mid) { /* display middle element */
            printf("%3d*", b[i]); /* mark middle value */
        } /* end else if */
        else { /* display other elements in subarray */
            printf("%3d ", b[i]);
        } /* end else */
    } /* end for */
    printf("\n"); /* start new line of output */
} /* end function printRow */

Enter a number between 0 and 28: 25
Subscripts:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14*</td>
<td>16</td>
<td>18</td>
<td>20</td>
<td>22</td>
<td>24</td>
<td>26</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>18</td>
<td>20</td>
<td>22*</td>
<td>24</td>
<td>26</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>26*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

25 not found

Enter a number between 0 and 28: 8
Subscripts:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14*</td>
<td>16</td>
<td>18</td>
<td>20</td>
<td>22</td>
<td>24</td>
<td>26</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>0 2</td>
<td>4</td>
<td>6*</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8 found in array element 4

Fig. 6.19  |  Binary search of a sorted array. (Part 3 of 4.)
Arrays in C can have multiple subscripts. A common use of multiple-subscripted arrays (also called multidimensional arrays) is to represent tables of values consisting of information arranged in rows and columns. To identify a particular table element, we must specify two subscripts: The first (by convention) identifies the element’s row and the second (by convention) identifies the element’s column. Tables or arrays that require two subscripts to identify a particular element are called double-subscripted arrays. Multiple-subscripted arrays can have more than two subscripts.

Figure 6.20 illustrates a double-subscripted array, \( a \). The array contains three rows and four columns, so it’s said to be a 3-by-4 array. In general, an array with \( m \) rows and \( n \) columns is called an \( m \)-by-\( n \) array.

Every element in array \( a \) is identified in Fig. 6.20 by an element name of the form \( a[i][j] \); \( a \) is the name of the array, and \( i \) and \( j \) are the subscripts that uniquely identify each element in \( a \). The names of the elements in the first row all have a first subscript of 0; the names of the elements in the fourth column all have a second subscript of 3.

**Common Programming Error 6.9**

Referencing a double-subscripted array element as \( a[x, y] \) instead of \( a[x][y] \). C interprets \( a[x, y] \) as \( a[y] \), and as such it does not cause a compilation error.
A multiple-subscripted array can be initialized when it’s defined, much like a single-subscripted array. For example, a double-subscripted array `int b[2][2]` could be defined and initialized with

```c
int b[2][2] = { { 1, 2 }, { 3, 4 } };
```

The values are grouped by row in braces. The values in the first set of braces initialize row 0 and the values in the second set of braces initialize row 1. So, the values 1 and 2 initialize elements `b[0][0]` and `b[0][1]`, respectively, and the values 3 and 4 initialize elements `b[1][0]` and `b[1][1]`, respectively. If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0. Thus,

```c
int b[2][2] = { { 1 }, { 3, 4 } };
```

would initialize `b[0][0]` to 1, `b[0][1]` to 0, `b[1][0]` to 3 and `b[1][1]` to 4. Figure 6.21 demonstrates defining and initializing double-subscripted arrays.

The program defines three arrays of two rows and three columns (six elements each). The definition of `array1` (line 11) provides six initializers in two sublists. The first sublist initializes the first row (i.e., row 0) of the array to the values 1, 2 and 3; and the second sublist initializes the second row (i.e., row 1) of the array to the values 4, 5 and 6.

```c
/* Fig. 6.21: fig06_21.c */
#include <stdio.h>

void printArray( const int a[][ 3 ] ); /* function prototype */

/* function main begins program execution */
int main( void )
{
  /* initialize array1, array2, array3 */
  int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
  int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
  int array3[ 2 ][ 3 ] = { 1, 2, 4 };

  printf( "Values in array1 by row are:\n" );
  printArray( array1 );

  printf( "Values in array2 by row are:\n" );
  printArray( array2 );

  printf( "Values in array3 by row are:\n" );
  printArray( array3 );
  return 0; /* indicates successful termination */
} /* end main */

/* function to output array with two rows and three columns */
void printArray( const int a[][ 3 ] )
{
  int i; /* row counter */
  int j; /* column counter */
```
If the braces around each sublist are removed from the `array1` initializer list, the compiler initializes the elements of the first row followed by the elements of the second row. The definition of `array2` (line 12) provides five initializers. The initializers are assigned to the first row, then the second row. Any elements that do not have an explicit initializer are initialized to zero automatically, so `array2[1][2]` is initialized to 0.

The definition of `array3` (line 13) provides three initializers in two sublists. The sublist for the first row explicitly initializes the first two elements of the first row to 1 and 2. The third element is initialized to zero. The sublist for the second row explicitly initializes the first element to 4. The last two elements are initialized to zero.

The program calls `printArray` (lines 27–43) to output each array's elements. The function definition specifies the array parameter as `const int a[][3]`. When we receive a single-subscripted array as a parameter, the array brackets are empty in the function's parameter list. The first subscript of a multiple-subscripted array is not required either, but all subsequent subscripts are required. The compiler uses these subscripts to determine the locations in memory of elements in multiple-subscripted arrays. All array elements are stored consecutively in memory regardless of the number of subscripts. In a double-subscripted array, the first row is stored in memory followed by the second row.

Providing the subscript values in a parameter declaration enables the compiler to tell the function how to locate an element in the array. In a double-subscripted array, each row is basically a single-subscripted array. To locate an element in a particular row, the compiler must know how many elements are in each row so that it can skip the proper number of memory locations when accessing the array. Thus, when accessing `a[1][2]` in our example, the compiler knows to skip the three elements of the first row to get to the second row (row 1). Then, the compiler accesses the third element of that row (element 2).

Many common array manipulations use for repetition statements. For example, the following statement sets all the elements in the third row of array `a` in Fig. 6.20 to zero:

```c
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```
We specified the third row, therefore we know that the first subscript is always 2 (again, 0 is the first row and 1 is the second). The loop varies only the second subscript (i.e., the column). The preceding for statement is equivalent to the assignment statements:

```c
for ( column = 0; column <= 3; column++ ) {
    a[ 2 ][ column ] = 0;
}
```

The following nested for statement determines the total of all the elements in array a.

```c
total = 0;
for ( row = 0; row <= 2; row++ ) {
    for ( column = 0; column <= 3; column++ ) {
        total += a[ row ][ column ];
    }
}
```

The for statement totals the elements of the array one row at a time. The outer for statement begins by setting row (i.e., the row subscript) to 0 so that the elements of the first row may be totaled by the inner for statement. The outer for statement then increments row to 1, so the elements of the second row can be totaled. Then, the outer for statement increments row to 2, so the elements of the third row can be totaled. The result is printed when the nested for statement terminates.

**Two-Dimensional Array Manipulations**

Figure 6.22 performs several other common array manipulations on 3-by-4 array studentGrades using for statements. Each row of the array represents a student and each column represents a grade on one of the four exams the students took during the semester. The array manipulations are performed by four functions. Function `minimum` (lines 43–62) determines the lowest grade of any student for the semester. Function `maximum` (lines 65–84) determines the highest grade of any student for the semester. Function `average` (lines 87–98) determines a particular student’s semester average. Function `printArray` (lines 101–120) outputs the double-subscripted array in a neat, tabular format.
/* function main begins program execution */
int main( void )
{
  int student; /* student counter */

  /* initialize student grades for three students (rows) */
  const int studentGrades[ STUDENTS ][ EXAMS ] =
  { { 77, 68, 86, 73 },
    { 96, 87, 89, 78 },
    { 70, 90, 86, 81 } };

  /* output array studentGrades */
  printf( "The array is:\n" );
  printArray( studentGrades, STUDENTS, EXAMS );

  /* determine smallest and largest grade values */
  printf( "\n\nLowest grade: %d
Highest grade: %d\n", minimum( studentGrades, STUDENTS, EXAMS ),
         maximum( studentGrades, STUDENTS, EXAMS ) );

  /* calculate average grade for each student */
  for ( student = 0; student < STUDENTS; student++ ) {
    printf( "The average grade for student %d is %.2f\n", student, average( studentGrades[ student ], EXAMS ) );
  } /* end for */

  return 0; /* indicates successful termination */
} /* end main */

/* Find the minimum grade */
int minimum( const int grades[][ EXAMS ], int pupils, int tests )
{
  int i; /* student counter */
  int j; /* exam counter */
  int lowGrade = 100; /* initialize to highest possible grade */

  /* loop through rows of grades */
  for ( i = 0; i < pupils; i++ ) {

    /* loop through columns of grades */
    for ( j = 0; j < tests; j++ ) {

      if ( grades[ i ][ j ] < lowGrade ) {
        lowGrade = grades[ i ][ j ];
      } /* end if */
    } /* end inner for */
  } /* end outer for */

  return lowGrade; /* return minimum grade */
} /* end function minimum */
/* Find the maximum grade */
int maximum( const int grades[][ EXAMS ], int pupils, int tests )
{
    int i; /* student counter */
    int j; /* exam counter */
    int highGrade = 0; /* initialize to lowest possible grade */

    /* loop through rows of grades */
    for ( i = 0; i < pupils; i++ ) {
        /* loop through columns of grades */
        for ( j = 0; j < tests; j++ ) {
            if ( grades[ i ][ j ] > highGrade ) {
                highGrade = grades[ i ][ j ];
            } /* end if */
        } /* end inner for */
    } /* end outer for */

    return highGrade; /* return maximum grade */
} /* end function maximum */

/* Determine the average grade for a particular student */
double average( const int setOfGrades[], int tests )
{
    int i; /* exam counter */
    int total = 0; /* sum of test grades */

    /* total all grades for one student */
    for ( i = 0; i < tests; i++ ) {
        total += setOfGrades[ i ];
    } /* end for */

    return ( double ) total / tests; /* average */
} /* end function average */

/* Print the array */
void printArray( const int grades[][ EXAMS ], int pupils, int tests )
{
    int i; /* student counter */
    int j; /* exam counter */

    /* output column heads */
    printf( " [0] [1] [2] [3]" );

    /* output grades in tabular format */
    for ( i = 0; i < pupils; i++ ) {
        /* output label for row */
        printf( "\nstudentGrades[%d] ", i );

Fig. 6.22 | Double-subscripted arrays example. (Part 3 of 4.)
Functions minimum, maximum and printArray each receive three arguments—the studentGrades array (called grades in each function), the number of students (rows of the array) and the number of exams (columns of the array). Each function loops through array grades using nested for statements. The following nested for statement is from the function minimum definition:

```c
/* loop through rows of grades */
for ( i = 0; i < pupils; i++ ) {
    /* loop through columns of grades */
    for ( j = 0; j < tests; j++ ) {
        if ( grades[ i ][ j ] < lowGrade ) {
            lowGrade = grades[ i ][ j ];
        } /* end if */
    } /* end inner for */
} /* end outer for */
```

The outer for statement begins by setting i (i.e., the row subscript) to 0 so that the elements of the first row (i.e., the grades of the first student) can be compared to variable lowGrade in the body of the inner for statement. The inner for statement loops through the four grades of a particular row and compares each grade to lowGrade. If a grade is less than lowGrade, lowGrade is set to that grade. The outer for statement then increments the row subscript to 1. The elements of the second row are compared to variable lowGrade. The outer for statement then increments the row subscript to 2. The elements of the third row are compared to variable lowGrade. When execution of the nested structure is complete, lowGrade contains the smallest grade in the double-subscripted array. Function maximum works similarly to function minimum.

Function average (line 87) takes two arguments—a single-subscripted array of test results for a particular student called setOfGrades and the number of test results in the array. When average is called, the first argument studentGrades[student] is passed. This causes the address of one row of the double-subscripted array to be passed to average.
The argument `studentGrades[1]` is the starting address of the second row of the array. Remember that a double-subscripted array is basically an array of single-subscripted arrays and that the name of a single-subscripted array is the address of the array in memory. Function `average` calculates the sum of the array elements, divides the total by the number of test results and returns the floating-point result.

**Summary**

**Section 6.1 Introduction**
- Arrays are data structures consisting of related data items of the same type.
- Arrays are “static” entities in that they remain the same size throughout program execution.

**Section 6.2 Arrays**
- An array is a group of memory locations related by the fact that they all have the same name and the same type.
- To refer to a particular location or element in the array, specify the name of the array and the position number of the particular element in the array.
- The first element in every array is the zeroth element. Thus, the first element of array `c` is referred to as `c[0]`, the second element of array `c` is referred to as `c[1]`, the seventh element of array `c` is referred to as `c[6]`, and, in general, the `ith` element of array `c` is referred to as `c[i - 1]`.
- Array names, like other variable names, can contain only letters, digits and underscores. Array names cannot begin with a digit.
- The position number contained within square brackets is more formally called a subscript. A subscript must be an integer or an integer expression.
- The brackets used to enclose the subscript of an array are actually considered to be an operator in C. They have the same level of precedence as the function call operator.

**Section 6.3 Defining Arrays**
- Arrays occupy space in memory. You specify the type of each element and the number of elements in the array so that the computer may reserve the appropriate amount of memory.
- An array of type `char` can be used to store a character string.

**Section 6.4 Array Examples**
- The elements of an array can be initialized when the array is defined by following the definition with an equals sign and braces, `{}`, containing a comma-separated list of initializers. If there are fewer initializers than elements in the array, the remaining elements are initialized to zero.
- The statement `int n[10] = {0};` explicitly initializes the first element to zero and initializes the remaining nine elements to zero because there are fewer initializers than there are elements in the array. It’s important to remember that automatic arrays are not automatically initialized to zero. You must at least initialize the first element to zero for the remaining elements to be automatically zeroed. This method of initializing the array elements to 0 is performed at compile time for static arrays and at runtime for automatic arrays.
- If the array size is omitted from a definition with an initializer list, the number of elements in the array will be the number of elements in the initializer list.
- The `#define` preprocessor directive can be used to define a symbolic constant—an identifier that is replaced with replacement text by the C preprocessor before the program is compiled. When the program is preprocessed, all occurrences of the symbolic constant are replaced with the replacement text. Using symbolic constants to specify array sizes makes programs more scalable.
• C has no array bounds checking to prevent a program from referring to an element that does not exist. Thus, an executing program can “walk off” the end of an array without warning. You should ensure that all array references remain within the bounds of the array.

• A string such as “hello” is really a static array of individual characters in C.

• A character array can be initialized using a string literal. In this case, the size of the array is determined by the compiler based on the length of the string.

• Every string contains a special string-termination character called the null character. The character constant representing the null character is ‘\0’.

• A character array representing a string should always be defined large enough to hold the number of characters in the string and the terminating null character.

• Character arrays also can be initialized with individual character constants in an initializer list.

• Because a string is really an array of characters, we can access individual characters in a string directly using array subscript notation.

• You can input a string directly into a character array from the keyboard using scanf and the conversion specifier %s. The name of the character array is passed to scanf without the preceding & used with nonstring variables. The & is normally used to provide scanf with a variable’s location in memory so that a value can be stored there. An array name is the address of the start of the array; therefore, the & is not necessary.

• Function scanf reads characters from the keyboard until the first white-space character is encountered—it does not check the array size. Thus, scanf can write beyond the end of the array.

• A character array representing a string can be output with printf and the %s conversion specifier. The characters of the string are printed until a terminating null character is encountered.

• A static local variable exists for the duration of the program but is only visible in the function body. We can apply static to a local array definition so that the array is not created and initialized each time the function is called and the array is not destroyed each time the function is exited in the program. This reduces program execution time, particularly for programs with frequently called functions that contain large arrays.

• Arrays that are static are automatically initialized once at compile time. If you do not explicitly initialize a static array, that array’s elements are initialized to zero by the compiler.

Section 6.5 Passing Arrays to Functions

• To pass an array argument to a function, specify the name of the array without any brackets.

• Unlike char arrays that contain strings, other array types do not have a special terminator. For this reason, the size of an array is passed to a function, so that the function can process the proper number of elements.

• C automatically passes arrays to functions by reference—the called functions can modify the element values in the callers’ original arrays. The name of the array evaluates to the address of the first element of the array. Because the starting address of the array is passed, the called function knows precisely where the array is stored. Therefore, when the called function modifies array elements in its function body, it’s modifying the actual elements of the array in their original memory locations.

• Although entire arrays are passed by reference, individual array elements are passed by value exactly as simple variables are.

• Such simple single pieces of data (such as individual ints, floats and chars) are called scalars.

• To pass an element of an array to a function, use the subscripted name of the array element as an argument in the function call.
For a function to receive an array through a function call, the function’s parameter list must specify that an array will be received. The size of the array is not required between the array brackets. If it’s included, the compiler checks that it’s greater than zero, then ignores it.

When an array parameter is preceded by the `const` qualifier, the elements of the array become constant in the function body, and any attempt to modify an element of the array in the function body results in a compile-time error.

Section 6.6 Sorting Arrays

- Sorting data (i.e., placing the data into a particular order such as ascending or descending) is one of the most important computing applications.
- One sorting technique is called the bubble sort or the sinking sort, because the smaller values gradually “bubble” their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array. The technique is to make several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or if the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.
- Because of the way the successive comparisons are made, a large value may move down the array many positions on a single pass, but a small value may move up only one position.
- The chief virtue of the bubble sort is that it’s easy to program. However, the bubble sort runs slowly. This becomes apparent when sorting large arrays.

Section 6.7 Case Study: Computing Mean, Median and Mode Using Arrays

- The mean is the arithmetic average of a set of values.
- The median is the “middle value” in a sorted set of values.
- The mode is the value that occurs most frequently in a set of values.

Section 6.8 Searching Arrays

- The process of finding a particular element of an array is called searching.
- The linear search compares each element of the array with the search key. Since the array is not in any particular order, it’s just as likely that the value will be found in the first element as in the last. On average, therefore, the search key will be compared with half the elements of the array.
- The linear searching method works well for small or unsorted arrays. For sorted arrays, the high-speed binary search technique can be used.
- The binary search algorithm eliminates from consideration one-half of the elements in a sorted array after each comparison. The algorithm locates the middle element of the array and compares it to the search key. If they are equal, the search key is found and the array subscript of that element is returned. If they are not equal, the problem is reduced to searching one-half of the array. If the search key is less than the middle element of the array, the first half of the array is searched, otherwise the second half of the array is searched. If the search key is not found in the specified subarray (piece of the original array), the algorithm is repeated on one-quarter of the original array. The search continues until the search key is equal to the middle element of a subarray, or until the subarray consists of one element that is not equal to the search key (i.e., the search key is not found).
- When using a binary search, the maximum number of comparisons required for any array can be determined by finding the first power of 2 greater than the number of array elements.

Section 6.9 Multiple-Subscripted Arrays

- A common use of multiple-subscripted arrays (also called multidimensional arrays) is to represent tables of values consisting of information arranged in rows and columns. To identify a par-
ticular table element, we must specify two subscripts: The first (by convention) identifies the element’s row and the second (by convention) identifies the element’s column.

- Tables or arrays that require two subscripts to identify a particular element are called double-subscripted arrays.
- Multiple-subscripted arrays can have more than two subscripts.
- A multiple-subscripted array can be initialized when it’s defined, much like a single-subscripted array. The values are grouped by row in braces. If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0.
- The first subscript of a multiple-subscripted array parameter declaration is not required, but all subsequent subscripts are required. The compiler uses these subscripts to determine the locations in memory of elements in multiple-subscripted arrays. All array elements are stored consecutively in memory regardless of the number of subscripts. In a double-subscripted array, the first row is stored in memory followed by the second row.
- Providing the subscript values in a parameter declaration enables the compiler to tell the function how to locate an element in the array. In a double-subscripted array, each row is basically a single-subscripted array. To locate an element in a particular row, the compiler must know how many elements are in each row so that it can skip the proper number of memory locations when accessing the array.

**Terminology**

- array 196
- binary search 223
- bubble sort 216
- const keyword 215
- double-subscripted array 229
- element 196
- index (or subscript) 196
- initializer 199
- key value 223
- linear search 223
- m-by-n array 229
- multidimensional array 229
- multiple-subscripted array 229
- name 196
- null character 207
- position number 196
- replacement text 202
- scalable 202
- scalar 212
- search key 223
- searching 223
- sinking sort 216
- subscript 196
- survey data analysis 218
- symbolic constant 201
- table 229
- value 196
- zeroth element 196

**Self-Review Exercises**

6.1  Answer each of the following:

a) Lists and tables of values are stored in ________.
b) An array’s elements are related by the fact that they have the same ________ and ________.
c) The number used to refer to a particular element of an array is called its ________.
d) A(n) ________ should be used to specify the size of an array because it makes the program more scalable.
e) The process of placing the elements of an array in order is called ________ the array.
f) Determining whether an array contains a certain key value is called ________ the array.
g) An array that uses two subscripts is referred to as a(n) ________ array.

6.2  State whether the following are true or false. If the answer is false, explain why.

a) An array can store many different types of values.
b) An array subscript can be of data type double.
c) If there are fewer initializers in an initializer list than the number of elements in the array, C automatically initializes the remaining elements to the last value in the list of initializers.

d) It’s an error if an initializer list contains more initializers than there are elements in the array.

e) An individual array element that is passed to a function as an argument of the form a[i] and modified in the called function will contain the modified value in the calling function.

6.3 Answer the following questions regarding an array called `fractions`.

a) Define a symbolic constant `SIZE` to be replaced with the replacement text 10.

b) Define an array with `SIZE` elements of type `double` and initialize the elements to 0.

c) Name the fourth element from the beginning of the array.

d) Refer to array element 4.

e) Assign the value 1.667 to array element nine.

f) Assign the value 3.333 to the seventh element of the array.

g) Print array elements 6 and 9 with two digits of precision to the right of the decimal point, and show the output that is displayed on the screen.

h) Print all the elements of the array, using a `for` repetition statement. Assume the integer variable `x` has been defined as a control variable for the loop. Show the output.

6.4 Write statements to accomplish the following:

a) Define `table` to be an integer array and to have 3 rows and 3 columns. Assume the symbolic constant `SIZE` has been defined to be 3.

b) How many elements does the array `table` contain? Print the total number of elements.

c) Use a `for` repetition statement to initialize each element of `table` to the sum of its subscripts. Assume the integer variables `x` and `y` are defined as control variables.

d) Print the values of each element of array `table`. Assume the array was initialized with the definition:

```c
int table[ SIZE ][ SIZE ] =
{ { 1, 8 }, { 2, 4, 6 }, { 5 } };
```

6.5 Find the error in each of the following program segments and correct the error.

a) `#define SIZE 100;`

b) `SIZE = 10;`

c) `Assume int b[ 10 ] = { 0 }, i;
for ( i = 0; i <= 10; i++ ) {
    b[ i ] = 1;
}`

d) `#include <stdio.h>;`

e) `Assume int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
a[ 1, 1 ] = 5;`

f) `#define VALUE = 120`

Answers to Self-Review Exercises

6.1 a) Arrays. b) Name, type. c) Subscript. d) Symbolic constant. e) Sorting. f) Searching. g) Double-subscripted.

6.2 a) False. An array can store only values of the same type.

b) False. An array subscript must be an integer or an integer expression.

c) False. C automatically initializes the remaining elements to zero.

d) True.

e) False. Individual elements of an array are passed by value. If the entire array is passed to a function, then any modifications will be reflected in the original.
6.3  a)  \#define SIZE 10
b)  double fractions[ SIZE ] = { 0.0 }; 
c)  fractions[ 3 ]
d)  fractions[ 4 ]
e)  fractions[ 9 ] = 1.667;
f)  fractions[ 6 ] = 3.333;
g)  printf("%.2f %.2f\n", fractions[ 6 ], fractions[ 9 ]); 
   Output: 3.33 1.67.
h)  for ( x = 0; x < SIZE; x++ ) {
     printf("fractions[%d] = %f\n", x, fractions[ x ] );
}
   Output:
   fractions[0] = 0.000000
   fractions[1] = 0.000000
   fractions[2] = 0.000000
   fractions[3] = 0.000000
   fractions[4] = 0.000000
   fractions[5] = 0.000000
   fractions[6] = 3.333000
   fractions[7] = 0.000000
   fractions[8] = 0.000000
   fractions[9] = 1.667000

6.4  a)  int table[ SIZE ][ SIZE ];

b)  Nine elements. printf("%d\n", SIZE * SIZE );

c)  for ( x = 0; x < SIZE; x++ ) {
    for ( y = 0; y < SIZE; y++ ) {
      table[ x ][ y ] = x + y;
    }
}

d)  for ( x = 0; x < SIZE; x++ ) {
    for ( y = 0; y < SIZE; y++ ) {
      printf("table[%d][%d] = %d\n", x, y, table[ x ][ y ] );
    }
}
   Output:
   table[0][0] = 1
   table[0][1] = 8
   table[0][2] = 0
   table[1][0] = 2
   table[1][1] = 4
   table[1][2] = 6
   table[2][0] = 5
   table[2][1] = 0
   table[2][2] = 0

6.5  a)  Error: Semicolon at end of \#define preprocessor directive.
   Correction: Eliminate colon.

b)  Error: Assigning a value to a symbolic constant using an assignment statement.
   Correction: Assign a value to the symbolic constant in a \#define preprocessor directive without using the assignment operator as in \#define SIZE 10.

c)  Error: Referencing an array element outside the bounds of the array (b[ 10 ]). 
   Correction: Change the final value of the control variable to 9.
d) Error: Semicolon at end of `#include` preprocessor directive. Correction: Eliminate semicolon.
e) Error: Array subscripting done incorrectly. Correction: Change the statement to `a[1][1] = 5;`
f) Error: Assigning a value to a symbolic constant using an assignment statement. Correction: Assign a value to the symbolic constant in a `#define` preprocessor directive without using the assignment operator as in `#define VALUE 120`.

**Exercises**

**6.6** Fill in the blanks in each of the following:

a) C stores lists of values in _______.
b) The elements of an array are related by the fact that they _______.
c) When referring to an array element, the position number contained within parentheses is called a(n) _______.
d) The names of the five elements of array `p` are ________, ________, ________, and ________.
e) The contents of a particular element of an array is called the ________ of that element.
f) Naming an array, stating its type and specifying the number of elements in the array is called ________ the array.
g) The process of placing the elements of an array into either ascending or descending order is called ________.
h) In a double-subscripted array, the first subscript (by convention) identifies the ________ of an element and the second subscript (by convention) identifies the ________ of an element.
i) An `m-by-n` array contains ________ rows, ________ columns and ________ elements.
j) The name of the element in row 3 and column 5 of array `d` is ________.

**6.7** State which of the following are true and which are false. If false, explain why.

a) To refer to a particular location or element within an array, we specify the name of the array and the value of the particular element.
b) An array definition reserves space for the array.
c) To indicate that 100 locations should be reserved for integer array `p`, write `p[100];`
d) A C program that initializes the elements of a 15-element array to zero must contain one `for` statement.
e) A C program that totals the elements of a double-subscripted array must contain nested `for` statements.
f) The mean, median and mode of the following set of values are 5, 6 and 7, respectively: 1, 2, 5, 6, 7, 7, 7.

**6.8** Write statements to accomplish each of the following:

a) Display the value of the seventh element of character array `f`.
b) Input a value into element 4 of single-subscripted floating-point array `b`.
c) Initialize each of the five elements of single-subscripted integer array `g` to 8.
d) Total the elements of floating-point array `c` of 100 elements.
e) Copy array `a` into the first portion of array `b`. Assume `double a[11], b[34];`
f) Determine and print the smallest and largest values contained in 99-element floating-point array `w`.

**6.9** Consider a 2-by-5 integer array `t`.

a) Write a definition for `t`. 
Exercises

243

b) How many rows does t have?
c) How many columns does t have?
d) How many elements does t have?
e) Write the names of all the elements in the second row of t.
f) Write the names of all the elements in the third column of t.
g) Write a single statement that sets the element of t in row 1 and column 2 to zero.
h) Write a series of statements that initialize each element of t to zero. Do not use a repetition structure.
i) Write a nested for statement that initializes each element of t to zero.
j) Write a statement that inputs the values for the elements of t from the terminal.
k) Write a series of statements that determine and print the smallest value in array t.
l) Write a statement that displays the elements of the first row of t.
m) Write a statement that totals the elements of the fourth column of t.
n) Write a series of statements that print the array t in tabular format. List the column subscripts as headings across the top and list the row subscripts at the left of each row.

6.10 (Sales Commissions) Use a single-subscripted array to solve the following problem. A company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9% of their gross sales for that week. For example, a salesperson who grosses $3000 in sales in a week receives $200 plus 9% of $3000, or a total of $470. Write a C program (using an array of counters) that determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson’s salary is truncated to an integer amount):

- $200–299
- $300–399
- $400–499
- $500–599
- $600–699
- $700–799
- $800–899
- $900–999
- $1000 and over

6.11 (Bubble Sort) The bubble sort presented in Fig. 6.15 is inefficient for large arrays. Make the following simple modifications to improve the performance of the bubble sort.

a) After the first pass, the largest number is guaranteed to be in the highest-numbered element of the array; after the second pass, the two highest numbers are “in place,” and so on. Instead of making nine comparisons on every pass, modify the bubble sort to make eight comparisons on the second pass, seven on the third pass and so on.
b) The data in the array may already be in the proper order or near-proper order, so why make nine passes if fewer will suffice? Modify the sort to check at the end of each pass whether any swaps have been made. If none has been made, then the data must already be in the proper order, so the program should terminate. If swaps have been made, then at least one more pass is needed.

6.12 Write single statements that perform each of the following single-subscripted array operations:

a) Initialize the 10 elements of integer array counts to zeros.
b) Add 1 to each of the 15 elements of integer array bonus.
c) Read the 12 values of floating-point array monthlyTemperatures from the keyboard.
d) Print the five values of integer array bestScores in column format.

6.13 Find the error(s) in each of the following statements:

a) Assume: char str[5];
   scanf( "%s", str ); /* User types hello */
b) Assume: int a[3];
   printf("$d %d $d\n", a[1], a[2], a[3]);
c) double f[3] = {1.1, 10.01, 100.001, 1000.0001};
d) Assume: double d[2][10];
d[1,9] = 2.345;

6.14 (Mean, Median and Mode Program Modifications) Modify the program of Fig. 6.16 so function mode is capable of handling a tie for the mode value. Also modify function median so the two middle elements are averaged in an array with an even number of elements.

6.15 (Duplicate Elimination) Use a single-subscripted array to solve the following problem. Read in 20 numbers, each of which is between 10 and 100, inclusive. As each number is read, print it only if it's not a duplicate of a number already read. Provide for the “worst case” in which all 20 numbers are different. Use the smallest possible array to solve this problem.

6.16 Label the elements of 3-by-5 double-subscripted array sales to indicate the order in which they are set to zero by the following program segment:

   for ( row = 0; row <= 2; row++ ) {
      for ( column = 0; column <= 4; column++ ) {
         sales[ row ][ column ] = 0;
      }
   }

6.17 What does the following program do?

```c
/* ex06_17.c */
/* What does this program do? */
#include <stdio.h>
#define SIZE 10
int whatIsThis( const int b[], int p ); /* function prototype */
/* function main begins program execution */
int main( void )
{
   int x; /* holds return value of function whatIsThis */
   /* initialize array a */
   int a[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   x = whatIsThis( a, SIZE );
   printf( "Result is %d\n", x );
   return 0; /* indicates successful termination */
} /* end main */
/* what does this function do? */
int whatIsThis( const int b[], int p )
{
   /* base case */
   if ( p == 1 ) {
      return b[ 0 ];
   } /* end if */
   else { /* recursion step */
      return b[ p - 1 ] + whatIsThis( b, p - 1 );
   } /* end else */
} /* end function whatIsThis */
```
6.18 What does the following program do?

```c
/* ex06_18.c */
/* What does this program do? */
#include <stdio.h>
#define SIZE 10

/* function prototype */
void someFunction( const int b[], int startIndex, int size );

/* function main begins program execution */
int main( void )
{
    int a[ SIZE ] = { 8, 3, 1, 2, 6, 0, 9, 7, 4, 5 }; /* initialize a */
    printf( "Answer is:\n" );
    someFunction( a, 0, SIZE );
    printf( "\n" );
    return 0; /* indicates successful termination */
}

/* What does this function do? */
void someFunction( const int b[], int startIndex, int size )
{
    if ( startIndex < size ) {
        someFunction( b, startIndex + 1, size );
        printf( "%d  ", b[ startIndex ] );
    } /* end if */
} /* end function someFunction */
```

6.19 (Dice Rolling) Write a program that simulates the rolling of two dice. The program should use rand to roll the first die, and should use rand again to roll the second die. The sum of the two values should then be calculated. [Note: Since each die can show an integer value from 1 to 6, then the sum of the two values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 the least frequent sums.] Figure 6.23 shows the 36 possible combinations of the two dice. Your program should roll the two dice 36,000 times. Use a single-subscribed array to tally the numbers of times each possible sum appears. Print the results in a tabular format. Also, determine if the totals are reasonable; i.e., there are six ways to roll a 7, so approximately one-sixth of all the rolls should be 7.

Fig. 6.23 | Dice rolling outcomes.
6.20 (Game of Craps) Write a program that runs 1000 games of craps (without human intervention) and answers each of the following questions:

a) How many games are won on the first roll, second roll, ..., twentieth roll and after the twentieth roll?

b) How many games are lost on the first roll, second roll, ..., twentieth roll and after the twentieth roll?

c) What are the chances of winning at craps? [Note: You should discover that craps is one of the fairest casino games. What do you suppose this means?]

d) What is the average length of a game of craps?

e) Do the chances of winning improve with the length of the game?

6.21 (Airline Reservations System) A small airline has just purchased a computer for its new automated reservations system. The president has asked you to program the new system. You’ll write a program to assign seats on each flight of the airline’s only plane (capacity: 10 seats).

Your program should display the following menu of alternatives:

Please type 1 for "first class"
Please type 2 for "economy"

If the person types 1, then your program should assign a seat in the first class section (seats 1–5). If the person types 2, then your program should assign a seat in the economy section (seats 6–10). Your program should then print a boarding pass indicating the person’s seat number and whether it’s in the first class or economy section of the plane.

Use a single-subscripted array to represent the seating chart of the plane. Initialize all the elements of the array to 0 to indicate that all seats are empty. As each seat is assigned, set the corresponding element of the array to 1 to indicate that the seat is no longer available.

Your program should, of course, never assign a seat that has already been assigned. When the first class section is full, your program should ask the person if it’s acceptable to be placed in the economy section (and vice versa). If yes, then make the appropriate seat assignment. If no, then print the message "Next flight leaves in 3 hours."

6.22 (Total Sales) Use a double-subscripted array to solve the following problem. A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each different type of product sold. Each slip contains:

a) The salesperson number

b) The product number

c) The total dollar value of that product sold that day

Thus, each salesperson passes in between 0 and 5 sales slips per day. Assume that the information from all of the slips for last month is available. Write a program that will read all this information for last month’s sales and summarize the total sales by salesperson by product. All totals should be stored in the double-subscripted array sales. After processing all the information for last month, print the results in tabular format with each of the columns representing a particular salesperson and each of the rows representing a particular product. Cross total each row to get the total sales of each product for last month; cross total each column to get the total sales by salesperson for last month. Your tabular printout should include these cross totals to the right of the totaled rows and to the bottom of the totaled columns.

6.23 (Turtle Graphics) The Logo language, which is particularly popular among personal computer users, made the concept of turtle graphics famous. Imagine a mechanical turtle that walks around the room under the control of a C program. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves; while the pen is up, the turtle moves about freely without writing anything. In this problem you’ll simulate the operation of the turtle and create a computerized sketchpad as well.
Exercises

247

Use a 50-by-50 array $\text{floor}$ which is initialized to zeros. Read commands from an array that contains them. Keep track of the current position of the turtle at all times and whether the pen is currently up or down. Assume that the turtle always starts at position 0, 0 of the floor with its pen up. The set of turtle commands your program must process are shown in Fig. 6.24. Suppose that the turtle is somewhere near the center of the floor. The following "program" would draw and print a 12-by-12 square:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

As the turtle moves with the pen down, set the appropriate elements of array $\text{floor}$ to 1s. When the 6 command (print) is given, wherever there is a 1 in the array, display an asterisk, or some other character you choose. Wherever there is a zero, display a blank. Write a program to implement the turtle graphics capabilities discussed here. Write several turtle graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle graphics language.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pen up</td>
</tr>
<tr>
<td>2</td>
<td>Pen down</td>
</tr>
<tr>
<td>3</td>
<td>Turn right</td>
</tr>
<tr>
<td>4</td>
<td>Turn left</td>
</tr>
<tr>
<td>5, 10</td>
<td>Move forward 10 spaces (or a number other than 10)</td>
</tr>
<tr>
<td>6</td>
<td>Print the 50-by-50 array</td>
</tr>
<tr>
<td>9</td>
<td>End of data (sentinel)</td>
</tr>
</tbody>
</table>

**Fig. 6.24** | Turtle commands.

6.24 (Knight’s Tour) One of the more interesting puzzlers for chess buffs is the Knight’s Tour problem, originally proposed by the mathematician Euler. The question is this: Can the chess piece called the knight move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in depth here.

The knight makes L-shaped moves (over two in one direction and then over one in a perpendicular direction). Thus, from a square in the middle of an empty chessboard, the knight can make eight different moves (numbered 0 through 7) as shown in Fig. 6.25.

a) Draw an 8-by-8 chessboard on a sheet of paper and attempt a Knight’s Tour by hand. Put a 1 in the first square you move to, a 2 in the second square, a 3 in the third, and so on. Before starting the tour, estimate how far you think you’ll get, remembering that a full tour consists of 64 moves. How far did you get? Were you close to the estimate?

b) Now let’s develop a program that will move the knight around a chessboard. The board itself is represented by an 8-by-8 double-subscripted array board. Each of the squares is initialized to zero. We describe each of the eight possible moves in terms of both their
horizontal and vertical components. For example, a move of type 0 as shown in Fig. 6.25 consists of moving two squares horizontally to the right and one square vertically upward. Move 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two single-subscripted arrays, horizontal and vertical, as follows:

\[
\text{horizontal}[0] = 2 \\
\text{horizontal}[1] = 1 \\
\text{horizontal}[2] = -1 \\
\text{horizontal}[3] = -2 \\
\text{horizontal}[4] = -2 \\
\text{horizontal}[5] = -1 \\
\text{horizontal}[6] = 1 \\
\text{horizontal}[7] = 2 \\
\text{vertical}[0] = -1 \\
\text{vertical}[1] = -2 \\
\text{vertical}[2] = -2 \\
\text{vertical}[3] = -1 \\
\text{vertical}[4] = 1 \\
\text{vertical}[5] = 2 \\
\text{vertical}[6] = 2 \\
\text{vertical}[7] = 1
\]

Let the variables currentRow and currentColumn indicate the row and column of the knight’s current position on the board. To make a move of type moveNumber, where moveNumber is between 0 and 7, your program uses the statements

\[
\text{currentRow } += \text{ vertical[moveNumber]}; \\
\text{currentColumn } += \text{ horizontal[moveNumber]};
\]

Keep a counter that varies from 1 to 64. Record the latest count in each square the knight moves to. Remember to test each potential move to see if the knight has already visited that square. And, of course, test every potential move to make sure that the knight does not land off the chessboard. Now write a program to move the knight around the chessboard. Run the program. How many moves did the knight make?

---

Fig. 6.25 | The eight possible moves of the knight.

c) After attempting to write and run a Knight’s Tour program, you have probably developed some valuable insights. We’ll use these to develop a heuristic (or strategy) for mov-
Exercises

249

ing the knight. Heuristics do not guarantee success, but a carefully developed heuristic greatly improves the chance of success. You may have observed that the outer squares are in some sense more troublesome than the squares nearer the center of the board. In fact, the most troublesome, or inaccessible, squares are the four corners.

Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to, so that when the board gets congested near the end of the tour, there will be a greater chance of success.

We may develop an “accessibility heuristic” by classifying each of the squares according to how accessible it is and always moving the knight to the square (within the knight’s L-shaped moves, of course) that is most inaccessible. We label a double-subscripted array accessibility with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, the center squares are therefore rated as 8s, the corner squares are rated as 2s, and the other squares have accessibility numbers of 3, 4, or 6 as follows:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Now write a version of the Knight’s Tour program using the accessibility heuristic. At any time, the knight should move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. [Note: As the knight moves around the chessboard, your program should reduce the accessibility numbers as more and more squares become occupied. In this way, at any given time during the tour, each available square’s accessibility number will remain equal to precisely the number of squares from which that square may be reached.] Run this version of your program. Did you get a full tour? Now modify the program to run 64 tours, one from each square of the chessboard. How many full tours did you get?

d) Write a version of the Knight’s Tour program which, when encountering a tie between two or more squares, decides what square to choose by looking ahead to those squares reachable from the “tied” squares. Your program should move to the square for which the next move would arrive at a square with the lowest accessibility number.

6.25 (Knight’s Tour: Brute-Force Approaches) In Exercise 6.24 we developed a solution to the Knight’s Tour problem. The approach used, called the “accessibility heuristic,” generates many solutions and executes efficiently.

As computers continue increasing in power, we’ll be able to solve many problems with sheer computer power and relatively unsophisticated algorithms. Let’s call this approach “brute-force” problem solving.

a) Use random number generation to enable the knight to walk around the chess board (in its legitimate L-shaped moves, of course) at random. Your program should run one tour and print the final chessboard. How far did the knight get?

b) Most likely, the preceding program produced a relatively short tour. Now modify your program to attempt 1000 tours. Use a single-subscripted array to keep track of the number of tours of each length. When your program finishes attempting the 1000 tours, it should print this information in neat tabular format. What was the best result?

c) Most likely, the preceding program gave you some “respectable” tours but no full tours. Now “pull all the stops out” and simply let your program run until it produces a full
tour. [Caution: This version of the program could run for hours on a powerful computer.] Once again, keep a table of the number of tours of each length and print this table when the first full tour is found. How many tours did your program attempt before producing a full tour? How much time did it take?

d) Compare the brute-force version of the Knight’s Tour with the accessibility heuristic version. Which required a more careful study of the problem? Which algorithm was more difficult to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility heuristic approach? Could we be certain (in advance) of obtaining a full tour with the brute-force approach? Argue the pros and cons of brute-force problem solving in general.

6.26 (Eight Queens) Another puzzler for chess buffs is the Eight Queens problem. Simply stated: Is it possible to place eight queens on an empty chessboard so that no queen is “attacking” any other—that is, so that no two queens are in the same row, the same column, or along the same diagonal? Use the kind of thinking developed in Exercise 6.24 to formulate a heuristic for solving the Eight Queens problem. Run your program. [Hint: It’s possible to assign a numeric value to each square of the chessboard indicating how many squares of an empty chessboard are “eliminated” once a queen is placed in that square. For example, each of the four corners would be assigned the value 22, as in Fig. 6.26.]

Once these “elimination numbers” are placed in all 64 squares, an appropriate heuristic might be: Place the next queen in the square with the smallest elimination number. Why is this strategy intuitively appealing?

6.27 (Eight Queens: Brute-Force Approaches) In this problem you’ll develop several brute-force approaches to solving the Eight Queens problem introduced in Exercise 6.26.

a) Solve the Eight Queens problem, using the random brute-force technique developed in Exercise 6.25.

b) Use an exhaustive technique (i.e., try all possible combinations of eight queens on the chessboard).

c) Why do you suppose the exhaustive brute-force approach may not be appropriate for solving the Eight Queens problem?

d) Compare and contrast the random brute-force and exhaustive brute-force approaches in general.

---

**Fig. 6.26** | The 22 squares eliminated by placing a queen in the upper-left corner.

6.28 (Duplicate Elimination) In Chapter 12, we explore the high-speed binary search tree data structure. One feature of a binary search tree is that duplicate values are discarded when insertions are made into the tree. This is referred to as duplicate elimination. Write a program that produces
20 random numbers between 1 and 20. The program should store all nonduplicate values in an array. Use the smallest possible array to accomplish this task.

6.29  *(Knight’s Tour: Closed Tour Test)* In the Knight’s Tour, a full tour is when the knight makes 64 moves touching each square of the chessboard once and only once. A closed tour occurs when the 64th move is one move away from the location in which the knight started the tour. Modify the Knight’s Tour program you wrote in Exercise 6.24 to test for a closed tour if a full tour has occurred.

6.30  *(The Sieve of Eratosthenes)* A prime integer is any integer greater than 1 that can be divided evenly only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It works as follows:

a) Create an array with all elements initialized to 1 (true). Array elements with prime subscripts will remain 1. All other array elements will eventually be set to zero.

b) Starting with array subscript 2 (subscript 1 is not prime), every time an array element is found whose value is 1, loop through the remainder of the array and set to zero every element whose subscript is a multiple of the subscript for the element with value 1. For array subscript 2, all elements beyond 2 in the array that are multiples of 2 will be set to zero (subscripts 4, 6, 8, 10, and so on.). For array subscript 3, all elements beyond 3 in the array that are multiples of 3 will be set to zero (subscripts 6, 9, 12, 15, and so on.).

When this process is complete, the array elements that are still set to 1 indicate that the subscript is a prime number. Write a program that uses an array of 1000 elements to determine and print the prime numbers between 1 and 999. Ignore element 0 of the array.

**Recursion Exercises**

6.31  *(Palindromes)* A palindrome is a string that is spelled the same way forward and backward. Some examples of palindromes are: “radar,” “able was i ere i saw elba,” and, if you ignore blanks, “a man a plan a canal panama.” Write a recursive function `testPalindrome` that returns 1 if the string stored in the array is a palindrome and 0 otherwise. The function should ignore spaces and punctuation in the string.

6.32  *(Linear Search)* Modify the program of Fig. 6.18 to use a recursive `linearSearch` function to perform the linear search of the array. The function should receive an integer array and the size of the array as arguments. If the search key is found, return the array subscript; otherwise, return –1.

6.33  *(Binary Search)* Modify the program of Fig. 6.19 to use a recursive `binarySearch` function to perform the binary search of the array. The function should receive an integer array and the starting subscript and ending subscript as arguments. If the search key is found, return the array subscript; otherwise, return –1.

6.34  *(Eight Queens)* Modify the Eight Queens program you created in Exercise 6.26 to solve the problem recursively.

6.35  *(Print an array)* Write a recursive function `printArray` that takes an array and the size of the array as arguments, prints the array, and returns nothing. The function should stop processing and return when it receives an array of size zero.

6.36  *(Print a string backward)* Write a recursive function `stringReverse` that takes a character array as an argument, prints it back to front and returns nothing. The function should stop processing and return when the terminating null character of the string is encountered.

6.37  *(Find the minimum value in an array)* Write a recursive function `recursiveMinimum` that takes an integer array and the array size as arguments and returns the smallest element of the array. The function should stop processing and return when it receives an array of one element.
Special Section: Sudoku

The game of Sudoku exploded in popularity worldwide in 2005. Almost every major newspaper now publishes a Sudoku puzzle daily. Handheld game players let you play anytime, anywhere and create puzzles on demand at various levels of difficulty. Be sure to check out our Sudoku Resource Center at www.deitel.com/sudoku for downloads, tutorials, books, e-books and more that will help you master the game. And not for the faint of heart—try fiendishly difficult Sudokus with tricky twists, a circular Sudoku and a variant of the puzzle with five interlocking grids. Subscribe to our free newsletter, the Deitel® Buzz Online, for notifications of updates to our Sudoku Resource Center and to other Deitel Resource Centers at www.deitel.com that provide games, puzzles and other interesting programming projects.

A completed Sudoku puzzle is a 9×9 grid (i.e., a two-dimensional array) in which the digits 1 through 9 appear once and only once in each row, each column and each of nine 3×3 grids. In the partially completed 9×9 grid of Fig. 6.27, row 1, column 1, and the 3×3 grid in the upper-left corner of the board each contain the digits 1 through 9 once and only once. We use C’s two-dimensional array row and column-numbering conventions, but we’re ignoring row 0 and column 0 in conformance with Sudoku community conventions.

The typical Sudoku puzzle provides many filled-in cells and many blanks, often arranged in a symmetrical pattern as is typical with crossword puzzles. The player’s task is to fill in the blanks to complete the puzzle. Some puzzles are easy to solve; some are quite difficult, requiring sophisticated solution strategies.

In Appendix D, Game Programming: Solving Sudoku, we’ll discuss various simple solution strategies, and suggest what to do when these fail. We’ll also present various approaches for programming Sudoku puzzle creators and solvers in C. Unfortunately, Standard C does not include graphics and GUI (graphical user interface) capabilities, so our representation of the board won’t be as elegant as we could make it in Java and other programming languages that support these capabilities. You may want to revisit your Sudoku programs after you read Appendix E, Game Programming with the Allegro C Library. Allegro, which is not part of Standard C, offers capabilities that will help you add graphics and even sounds to your Sudoku programs.

![Fig. 6.27](image_url) | Partially completed 9×9 Sudoku grid. Note the nine 3×3 grids.
Addresses are given to us to conceal our whereabouts.
—Saki (H. H. Munro)

By indirection find direction out.
—William Shakespeare

Many things, having full reference
To one consent, may work contrariously.
—William Shakespeare

You will find it a very good practice always to verify your references, sir!
—Dr. Routh

Objectives
In this chapter, you’ll learn:

- Pointers and pointer operators.
- To use pointers to pass arguments to functions by reference.
- The close relationships among pointers, arrays and strings.
- To use pointers to functions.
- To define and use arrays of strings.
7.1 Introduction

In this chapter, we discuss one of the most powerful features of the C programming language, the **pointer**. Pointers are among C’s most difficult capabilities to master. Pointers enable programs to simulate call-by-reference and to create and manipulate dynamic data structures, i.e., data structures that can grow and shrink at execution time, such as linked lists, queues, stacks and trees. This chapter explains basic pointer concepts. Chapter 10 examines the use of pointers with structures. Chapter 12 introduces dynamic memory management techniques and presents examples of creating and using dynamic data structures.

7.2 Pointer Variable Definitions and Initialization

Pointers are variables whose values are memory addresses. Normally, a variable directly contains a specific value. A pointer, on the other hand, contains an address of a variable that contains a specific value. In this sense, a variable name **directly** references a value, and a pointer **indirectly** references a value (Fig. 7.1). Referencing a value through a pointer is called **indirection**.

---

1. Pointers and pointer-based entities such as arrays and strings, when misused intentionally or accidentally, can lead to errors and security breaches. See our Secure C Programming Resource Center ([www.deitel.com/SecureC/](http://www.deitel.com/SecureC/)) for articles, books, white papers and forums on this important topic.
Pointers, like all variables, must be defined before they can be used. The definition

```c
int *countPtr, count;
```

specifies that variable countPtr is of type int * (i.e., a pointer to an integer) and is read, “countPtr is a pointer to int” or “countPtr points to an object of type int.” Also, the variable count is defined to be an int, not a pointer to an int. The * only applies to countPtr in the definition. When * is used in this manner in a definition, it indicates that the variable being defined is a pointer. Pointers can be defined to point to objects of any type.

### Common Programming Error 7.1
*The asterisk (*) notation used to declare pointer variables does not distribute to all variable names in a declaration. Each pointer must be declared with the * prefixed to the name; e.g., if you wish to declare xPtr and yPtr as int pointers, use int *xPtr, *yPtr;.*

### Common Programming Error 7.2
*Include the letters ptr in pointer variable names to make it clear that these variables are pointers and thus need to be handled appropriately.*

Pointers should be initialized either when they’re defined or in an assignment statement. A pointer may be initialized to NULL, 0 or an address. A pointer with the value NULL points to nothing. NULL is a symbolic constant defined in the `<stddef.h>` header (and several other headers, such as `<stdio.h>`). Initializing a pointer to 0 is equivalent to initializing a pointer to NULL, but NULL is preferred. When 0 is assigned, it’s first converted to a pointer of the appropriate type. The value 0 is the only integer value that can be assigned directly to a pointer variable. Assigning a variable’s address to a pointer is discussed in Section 7.3.

### Error-Prevention Tip 7.1
*Initialize pointers to prevent unexpected results.*

## 7.3 Pointer Operators

The &, or *address operator*, is a unary operator that returns the address of its operand. For example, assuming the definitions

```c
int y = 5;
int *yPtr;
```

the statement

```c
yPtr = &y;
```

assigns the address of the variable y to pointer variable yPtr. Variable yPtr is then said to “point to” y. Figure 7.2 shows a schematic representation of memory after the preceding assignment is executed.

![Graphical representation of a pointer pointing to an integer variable in memory.](image)
Figure 7.3 shows the representation of the pointer in memory, assuming that integer variable y is stored at location 600000, and pointer variable yPtr is stored at location 500000. The operand of the address operator must be a variable; the address operator cannot be applied to constants, to expressions or to variables declared with the storage-class register.

The unary * operator, commonly referred to as the indirection operator or dereferencing operator, returns the value of the object to which its operand (i.e., a pointer) points. For example, the statement

```c
printf( "%d", *yPtr );
```

prints the value of variable y, namely 5. Using * in this manner is called dereferencing a pointer.

**Common Programming Error 7.3**

Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory is an error. This could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion with incorrect results.

Figure 7.4 demonstrates the pointer operators & and *. The printf conversion specifier %p outputs the memory location as a hexadecimal integer on most platforms. (See Appendix C, Number Systems, for more information on hexadecimal integers.) Notice that the address of a and the value of aPtr are identical in the output, thus confirming that the address of a is indeed assigned to the pointer variable aPtr (line 11). The & and * operators are complements of one another—when they’re both applied consecutively to aPtr in either order (line 21), the same result is printed. Figure 7.5 lists the precedence and associativity of the operators introduced to this point.

```c
/* Fig. 7.4: fig07_04.c */
#include <stdio.h>

int main( void )
{
    int a; /* a is an integer */
    int *aPtr; /* aPtr is a pointer to an integer */

    a = 7;
    aPtr = &a; /* aPtr set to address of a */
}
```

Fig. 7.4 | Using the & and * pointer operators. (Part 1 of 2.)
7.4 Passing Arguments to Functions by Reference

There are two ways to pass arguments to a function—call-by-value and call-by-reference. All arguments in C are passed by value. As we saw in Chapter 5, return may be used to return one value from a called function to a caller (or to return control from a called function without passing back a value). Many functions require the capability to modify one or more variables in the caller or to pass a pointer to a large data object to avoid the over-
head of passing the object by value (which incurs the overhead of making a copy of the object). For these purposes, C provides the capabilities for simulating call-by-reference.

In C, you use pointers and the indirection operator to simulate call-by-reference. When calling a function with arguments that should be modified, the addresses of the arguments are passed. This is normally accomplished by applying the address operator (&) to the variable (in the caller) whose value will be modified. As we saw in Chapter 6, arrays are not passed using operator & because C automatically passes the starting location in memory of the array (the name of an array is equivalent to &arrayName[0]). When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to modify the value at that location in the caller’s memory.

The programs in Fig. 7.6 and Fig. 7.7 present two versions of a function that cubes an integer—cubeByValue and cubeByReference. Figure 7.6 passes the variable number to function cubeByValue using call-by-value (line 14). The cubeByValue function cubes its argument and passes the new value back to main using a return statement. The new value is assigned to number in main (line 14).

```c
int cubeByValue( int n ); /* prototype */

int main( void )
{
    int number = 5; /* initialize number */
    printf( "The original value of number is %d", number );
    /* pass number by value to cubeByValue */
    number = cubeByValue( number );
    printf( "The new value of number is %d\n", number );
    return 0; /* indicates successful termination */
}

int cubeByValue( int n )
{
    return n * n * n; /* cube local variable n and return result */
}
```

The original value of number is 5
The new value of number is 125

**Fig. 7.6** Cube a variable using call-by-value.

Figure 7.7 passes the variable number using call-by-reference (line 15)—the address of number is passed—to function cubeByReference. Function cubeByReference takes as a parameter a pointer to an int called nPtr (line 22). The function dereferences the pointer and cubes the value to which nPtr points (line 24), then assigns the result to *nPtr (which
is really `number` in `main`), thus changing the value of `number` in `main`. Figure 7.8 and Fig. 7.9 analyze graphically the programs in Fig. 7.6 and Fig. 7.7, respectively.

A function receiving an address as an argument must define a pointer parameter to receive the address. For example, in Fig. 7.7 the header for function `cubeByReference` (line 22) is:

```c
void cubeByReference(int *nPtr); /* prototype */
```

The header specifies that `cubeByReference` receives the address of an integer variable as an argument, stores the address locally in `nPtr` and does not return a value.

The function prototype for `cubeByReference` contains `int *` in parentheses. As with other variable types, it’s not necessary to include names of pointers in function prototypes. Names included for documentation purposes are ignored by the C compiler.

In the function header and in the prototype for a function that expects a single-subscripted array as an argument, the pointer notation in the parameter list of function `cubeByReference` may be used. The compiler does not differentiate between a function that receives a pointer and a function that receives a single-subscripted array. This, of course, means that the function must “know” when it’s receiving an array or simply a single variable for which it is to perform call by reference. When the compiler encounters
a function parameter for a single-subscripted array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b`. The two forms are interchangeable.
7.5 Using the const Qualifier with Pointers

The const qualifier enables you to inform the compiler that the value of a particular variable should not be modified. The const qualifier did not exist in early versions of C; it was added to the language by the ANSI C committee.

Software Engineering Observation 7.1

The const qualifier can be used to enforce the principle of least privilege. Using the principle of least privilege to properly design software reduces debugging time and improper side effects, making a program easier to modify and maintain.

Portability Tip 7.1

Although const is well defined in Standard C, some compilers do not enforce it.
Over the years, a large base of legacy code was written in early versions of C that did not use `const` because it was not available. For this reason, there are significant opportunities for improvement in the reengineering of old C code.

Six possibilities exist for using (or not using) `const` with function parameters—two with call-by-value parameter passing and four with call-by-reference parameter passing. How do you choose one of the six possibilities? Let the principle of least privilege be your guide. Always award a function enough access to the data in its parameters to accomplish its specified task, but no more.

In Chapter 5, we explained that all calls in C are call-by-value—a copy of the argument in the function call is made and passed to the function. If the copy is modified in the function, the original value in the caller does not change. In many cases, a value passed to a function is modified so the function can accomplish its task. However, in some instances, the value should not be altered in the called function, even though it manipulates only a copy of the original value.

Consider a function that takes a single-subscripted array and its size as arguments and prints the array. Such a function should loop through the array and output each array element individually. The size of the array is used in the function body to determine the high subscript of the array, so the loop can terminate when the printing is completed. Neither the size of the array nor its contents should change in the function body.

**Error-Prevention Tip 7.3**

If a variable does not (or should not) change in the body of a function to which it’s passed, the variable should be declared `const` to ensure that it’s not accidentally modified.

If an attempt is made to modify a value that is declared `const`, the compiler catches it and issues either a warning or an error, depending on the particular compiler.

**Software Engineering Observation 7.2**

Only one value in a calling function can be altered when using call-by-value. That value must be assigned from the return value of the function to a variable in the caller. To modify multiple variables from a calling function in a called function, use call-by-reference.

**Error-Prevention Tip 7.4**

Before using a function, check its function prototype to determine if the function is able to modify the values passed to it.

**Common Programming Error 7.4**

Being unaware that a function is expecting pointers as arguments for call-by-reference and passing arguments call-by-value. Some compilers take the values assuming they’re pointers and dereference the values as pointers. At runtime, memory-access violations or segmentation faults are often generated. Other compilers catch the mismatch in types between arguments and parameters and generate error messages.

There are four ways to pass a pointer to a function: a non-constant pointer to non-constant data, a constant pointer to nonconstant data, a non-constant pointer to constant data, and a constant pointer to constant data. Each of the four combinations provides different access privileges. These are discussed in the next several examples.
Converting a String to Uppercase Using a Non-Constant Pointer to Non-Constant Data

The highest level of data access is granted by a non-constant pointer to non-constant data. In this case, the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data items. A declaration for a non-constant pointer to non-constant data does not include `const`. Such a pointer might be used to receive a string as an argument to a function that uses `pointer arithmetic` to process (and possibly modify) each character in the string. Function `convertToUppercase` of Fig. 7.10 declares its parameter, a non-constant pointer to non-constant data called `sPtr` (char *sPtr), in line 21. The function processes the array string (pointed to by `sPtr`) one character at a time using pointer arithmetic. C standard library function `islower` (called in line 25) tests the character contents of the address pointed to by `sPtr`. If a character is in the range a to z, `islower` returns true and C standard library function `toupper` (line 26) is called to convert the character to its corresponding uppercase letter; otherwise, `islower` returns false and the next character in the string is processed. Line 29 moves the pointer to the next character in the string. Pointer arithmetic will be discussed in more detail in Section 7.8.

```c
/* Fig. 7.10: fig07_10.c */
/* Converting a string to uppercase using a non-constant pointer to non-constant data */
#include <stdio.h>
#include <ctype.h>

void convertToUppercase( char *sPtr ); /* prototype */

int main( void )
{
    char string[] = "characters and $32.98"; /* initialize char array */
    printf( "The string before conversion is: %s", string );
    convertToUppercase( string );
    printf( "The string after conversion is: %s\n", string );
    return 0; /* indicates successful termination */
}

/* convert string to uppercase letters */
void convertToUppercase( char *sPtr )
{
    while ( *sPtr != '\0' ) { /* current character is not '\0' */
        if ( islower( *sPtr ) ) { /* if character is lowercase, */
            *sPtr = toupper( *sPtr ); /* convert to uppercase */
        } /* end if */
        ++sPtr; /* move sPtr to the next character */
    } /* end while */
} /* end function convertToUppercase */
```

Fig. 7.10 | Converting a string to uppercase using a non-constant pointer to non-constant data.
(Part 1 of 2.)
Chapter 7  C Pointers

Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data

A non-constant pointer to constant data can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified. Such a pointer might be used to receive an array argument to a function that will process each element without modifying the data. For example, function printCharacters (Fig. 7.11) declares parameter sPtr to be of type const char * (line 22). The declaration is read from right to left as “sPtr is a pointer to a character constant.” The function uses a for statement to output each character in the string until the null character is encountered. After each character is printed, pointer sPtr is incremented to point to the next character in the string.

```
/* Fig. 7.11: fig07_11.c */

#include <stdio.h>

void printCharacters( const char *sPtr );

int main( void )
{
    /* initialize char array */
    char string[] = "print characters of a string";

    printf( "The string is:\n" );
    printCharacters( string );
    printf( "\n" );
    return 0; /* indicates successful termination */
}

/* sPtr cannot modify the character to which it points,
   i.e., sPtr is a "read-only" pointer */
void printCharacters( const char *sPtr )
{
    /* loop through entire string */
    for ( ; *sPtr != '\0'; sPtr++ ) { /* no initialization */
        printf( "%c", *sPtr );
    } /* end for */
}
```

The string is:
print characters of a string
7.5 Using the \const\ Qualifier with Pointers

Figure 7.12 illustrates the attempt to compile a function that receives a non-constant pointer (xPtr) to constant data. This function attempts to modify the data pointed to by xPtr in line 20—which results in a compilation error. [Note: The actual error message you see will be compiler specific.]

As we know, arrays are aggregate data types that store related data items of the same type under one name. In Chapter 10, we'll discuss another form of aggregate data type called a structure (sometimes called a record in other languages). A structure is capable of storing related data items of different data types under one name (e.g., storing information about each employee of a company). When a function is called with an array as an argument, the array is automatically passed to the function by reference. However, structures are always passed by value—a copy of the entire structure is passed. This requires the execution-time overhead of making a copy of each data item in the structure and storing it on the computer’s function call stack. When structure data must be passed to a function, we can use pointers to constant data to get the performance of call-by-reference and the protection of call-by-value. When a pointer to a structure is passed, only a copy of the address at which the structure is stored must be made. On a machine with 4-byte addresses, a copy of 4 bytes of memory is made rather than a copy of possibly hundreds or thousands of bytes of the structure.
Using pointers to constant data in this manner is an example of a \textit{time/space trade-off}. If memory is low and execution efficiency is a concern, use pointers. If memory is in abundance and efficiency is not a major concern, pass data by value to enforce the principle of least privilege. Remember that some systems do not enforce const well, so call-by-value is still the best way to prevent data from being modified.

\textbf{Attempting to Modify a Constant Pointer to Non-Constant Data}

A constant pointer to non-constant data always points to the same memory location, and the data at that location can be modified through the pointer. This is the default for an array name. An array name is a constant pointer to the beginning of the array. All data in the array can be accessed and changed by using the array name and array subscripting. A constant pointer to non-constant data can be used to receive an array as an argument to a function that accesses array elements using only array subscript notation. Pointers that are declared \texttt{const} must be initialized when they’re defined (if the pointer is a function parameter, it’s initialized with a pointer that is passed to the function). Figure 7.13 attempts to modify a constant pointer. Pointer \texttt{ptr} is defined in line 12 to be of type \texttt{int * const}. The definition is read from right to left as “\texttt{ptr} is a constant pointer to an integer.” The pointer is initialized (line 12) with the address of integer variable \texttt{x}. The program attempts to assign the address of \texttt{y} to \texttt{ptr} (line 15), but the compiler generates an error message.

\begin{verbatim}
/* Fig. 7.13: fig07_13.c
   Attempting to modify a constant pointer to non-constant data */
#include <stdio.h>
int main( void )
{
    int x; /* define x */
    int y; /* define y */

    /* ptr is a constant pointer to an integer that can be modified
       through ptr, but ptr always points to the same memory location */
    int * const ptr = &x;

    *ptr = 7; /* allowed: *ptr is not const */
    ptr = &y; /* error: ptr is const; cannot assign new address */
    return 0; /* indicates successful termination */
} /* end main */
\end{verbatim}

Compiling...
FIG07_13.c
C:\examples\ch07\FIG07_13.c(15) : error C2166: l-value specifies const object
Error executing cl.exe.

FIG07_13.exe - 1 error(s), 0 warning(s)
**Attempting to Modify a Constant Pointer to Constant Data**

The least access privilege is granted by a constant pointer to constant data. Such a pointer always points to the same memory location, and the data at that memory location cannot be modified. This is how an array should be passed to a function that only looks at the array using array subscript notation and does not modify the array. Figure 7.14 defines pointer variable `ptr` (line 13) to be of type `const int *const`, which is read from right to left as “`ptr` is a constant pointer to an integer constant.” The figure shows the error messages generated when an attempt is made to modify the data to which `ptr` points (line 16) and when an attempt is made to modify the address stored in the pointer variable (line 17).

```c
/* Fig. 7.14: fig07_14.c */
#include <stdio.h>

int main( void )
{
    int x = 5; /* initialize x */
    int y; /* define y */

    /* ptr is a constant pointer to a constant integer. ptr always
     * points to the same location; the integer at that location
     * cannot be modified */
    const int *const ptr = &x;

    printf( "\n", *ptr );
    *ptr = 7; /* error: *ptr is const; cannot assign new value */
    ptr = &y; /* error: ptr is const; cannot assign new address */
    return 0; /* indicates successful termination */
}
```

**Fig. 7.14** | Attempting to modify a constant pointer to constant data.

---

**7.6 Bubble Sort Using Call-by-Reference**

Let’s improve the bubble sort program of Fig. 6.15 to use two functions—`bubbleSort` and `swap`. Function `bubbleSort` sorts the array. It calls function `swap` (line 51) to exchange the array elements `array[j]` and `array[j + 1]` (see Fig.). Remember that C enforces information hiding between functions, so `swap` does not have access to individual array elements in `bubbleSort`. Because `bubbleSort` wants `swap` to have access to the array elements to be swapped, `bubbleSort` passes each of these elements call-by-reference to `swap`—the address of each array element is passed explicitly. Although entire arrays are automatically passed by reference, individual array elements are scalars and are ordinarily passed by val-
ue. Therefore, `bubbleSort` uses the address operator (`&`) on each of the array elements in the `swap` call (line 51) to effect call-by-reference as follows:

```
swap( &array[ j ], &array[ j + 1 ] );
```

Function `swap` receives `&array[j]` in pointer variable `element1Ptr` (line 59). Even though `swap`—because of information hiding—is not allowed to know the name `array[j]`, `swap` may use `*element1Ptr` as a synonym for `array[j]`—when `swap` references `*element1Ptr`, it’s actually referencing `array[j]` in `bubbleSort`. Similarly, when `swap` references `*element2Ptr`, it’s actually referencing `array[j + 1]` in `bubbleSort`. Even though `swap` is not allowed to say

```
hold = array[ j ];
array[ j ] = array[ j + 1 ];
array[ j + 1 ] = hold;
```

precisely the same effect is achieved by lines 61 through 63:

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

---

```c
/* Fig. 7.15: fig07_15.c
This program puts values into an array, sorts the values into
ascending order, and prints the resulting array. */
#include <stdio.h>
#define SIZE 10

void bubbleSort( int * const array, const int size ); /* prototype */

int main( void )
{
    /* initialize array a */
    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

    int i; /* counter */

    printf( "Data items in original order\n" );

    /* loop through array a */
    for ( i = 0; i < SIZE; i++ ) {
        printf( "%4d", a[ i ] );
    } /* end for */

    bubbleSort( a, SIZE ); /* sort the array */

    printf( "\nData items in ascending order\n" );

    /* loop through array a */
    for ( i = 0; i < SIZE; i++ ) {
        printf( "%4d", a[ i ] );
    } /* end for */
```
Several features of function bubbleSort should be noted. The function header (line 37) declares array as int * const array rather than int array[] to indicate that bubbleSort receives a single-subscripted array as an argument (again, these notations are interchangeable). Parameter size is declared const to enforce the principle of least privilege. Although parameter size receives a copy of a value in main, and modifying the copy cannot change the value in main, bubbleSort does not need to alter size to accomplish its task. The size of the array remains fixed during the execution of function bubbleSort. Therefore, size is declared const to ensure that it’s not modified. If the size of the array is modified during the sorting process, the sorting algorithm might not run correctly.

The prototype for function swap (line 39) is included in the body of function bubbleSort because bubbleSort is the only function that calls swap. Placing the prototype in
bubbleSort restricts proper calls of swap to those made from bubbleSort. Other functions that attempt to call swap do not have access to a proper function prototype, so the compiler generates one automatically. This normally results in a prototype that does not match the function header (and generates a compilation warning or error) because the compiler assumes int for the return type and the parameter types.

Function bubbleSort receives the size of the array as a parameter (line 37). The function must know the size of the array to sort the array. When an array is passed to a function, the memory address of the first element of the array is received by the function. The address, of course, does not convey the number of elements in the array. Therefore, you must pass to the function the array size. [Note: Another common practice is to pass a pointer to the beginning of the array and a pointer to the location just beyond the end of the array. The difference of the two pointers is the length of the array and the resulting code is simpler.]

In the program, the size of the array is explicitly passed to function bubbleSort. There are two main benefits to this approach—software reusability and proper software engineering. By defining the function to receive the array size as an argument, we enable the function to be used by any program that sorts single-subscripted integer arrays of any size.

We could have stored the array’s size in a global variable that is accessible to the entire program. This would be more efficient, because a copy of the size is not made to pass to the function. However, other programs that require an integer array-sorting capability may not have the same global variable, so the function cannot be used in those programs.

The size of the array could have been programmed directly into the function. This restricts the use of the function to an array of a specific size and significantly reduces its reusability. Only programs processing single-subscripted integer arrays of the specific size coded into the function can use the function.

### 7.7 sizeof Operator

C provides the special unary operator sizeof to determine the size in bytes of an array (or any other data type) during program compilation. When applied to the name of an array as in Fig. 7.16 (line 14), the sizeof operator returns the total number of bytes in the array.
as an integer. Variables of type float are normally stored in 4 bytes of memory, and array is defined to have 20 elements. Therefore, there are a total of 80 bytes in array.

**Performance Tip 7.2**

sizeof is a compile-time operator, so it does not incur any execution-time overhead.

```c
/* Fig. 7.16: fig07_16.c */
Applying sizeof to an array name returns the number of bytes in the array. */
#include <stdio.h>

size_t getSize(); /* prototype */

int main(void)
{
    float array[20]; /* create array */

    printf("The number of bytes in the array is %d\n", sizeof( array ), getSize( array ));
    return 0; /* indicates successful termination */
} /* end main */

/* return size of ptr */
size_t getSize( float *ptr )
{
    return sizeof( ptr );
} /* end function getSize */
```

The number of bytes in the array is 80
The number of bytes returned by getSize is 4

**Fig. 7.16**  Applying sizeof to an array name returns the number of bytes in the array.

The number of elements in an array also can be determined with sizeof. For example, consider the following array definition:

```c
double real[22];
```

Variables of type double normally are stored in 8 bytes of memory. Thus, array real contains a total of 176 bytes. To determine the number of elements in the array, the following expression can be used:

```c
sizeof( real ) / sizeof( real[0] )
```

The expression determines the number of bytes in array real and divides that value by the number of bytes used in memory to store the first element of array real (a double value).

Function getSize returns type size_t. Type size_t is a type defined by the C standard as the integral type (unsigned or unsigned long) of the value returned by operator sizeof. Type size_t is defined in header <stddef.h> (which is included by several headers, such as <stdio.h>). [Note: If you attempt to compile Fig. 7.16 and receive errors,
simply include `<stddef.h>` in your program.] Figure 7.17 calculates the number of bytes used to store each of the standard data types. The results could be different between computers.

Portability Tip 7.2

The number of bytes used to store a particular data type may vary between systems. When writing programs that depend on data type sizes and that will run on several computer systems, use `sizeof` to determine the number of bytes used to store the data types.

```c
/* Fig. 7.17: fig07_17.c
Demonstrating the sizeof operator */
#include <stdio.h>

int main( void )
{
    char c;
    short s;
    int i;
    long l;
    float f;
    double d;
    long double ld;
    int array[20]; /* create array of 20 int elements */
    int *ptr = array; /* create pointer to array */
    printf("     sizeof c = %d	sizeof(char)  = %d"
           "     sizeof s = %d	sizeof(short) = %d"
           "     sizeof i = %d	sizeof(int) = %d"
           "     sizeof l = %d	sizeof(long) = %d"
           "     sizeof f = %d	sizeof(float) = %d"
           "     sizeof d = %d	sizeof(double) = %d"
           "     sizeof ld = %d	sizeof(long double) = %d"
           " sizeof array = %d"
           " sizeof ptr = %d\n",
           sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
           sizeof( int ), sizeof l, sizeof( long ), sizeof f,
           sizeof( float ), sizeof d, sizeof( double ), sizeof ld,
           sizeof( long double ), sizeof array, sizeof ptr);
    return 0; /* indicates successful termination */
}
/* end main */
```

Fig. 7.17 | Using operator `sizeof` to determine standard data type sizes.
Operator `sizeof` can be applied to any variable name, type or value (including the value of an expression). When applied to a variable name (that is not an array name) or a constant, the number of bytes used to store the specific type of variable or constant is returned. The parentheses used with `sizeof` are required if a type name with two words is supplied as its operand (such as `long double` or `unsigned short`). Omitting the parentheses in this case results in a syntax error. The parentheses are not required if a variable name or a one-word type name is supplied as its operand, but they can still be included without causing an error.

7.8 Pointer Expressions and Pointer Arithmetic

Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions. However, not all the operators normally used in these expressions are valid in conjunction with pointer variables. This section describes the operators that can have pointers as operands, and how these operators are used.

A limited set of arithmetic operations may be performed on pointers. A pointer may be incremented (++) or decremented (--), an integer may be added to a pointer (+ or +=), an integer may be subtracted from a pointer (- or -=) and one pointer may be subtracted from another.

Assume that array `int v[5]` has been defined and its first element is at location 3000 in memory. Assume pointer `vPtr` has been initialized to point to `v[0]`—i.e., the value of `vPtr` is 3000. Figure 7.18 illustrates this situation for a machine with 4-byte integers. Variable `vPtr` can be initialized to point to array `v` with either of the statements

```
    vPtr = v;
    vPtr = &v[0];
```

Portability Tip 7.3

Most computers today have 2-byte or 4-byte integers. Some of the newer machines use 8-byte integers. Because the results of pointer arithmetic depend on the size of the objects a pointer points to, pointer arithmetic is machine dependent.

In conventional arithmetic, 3000 + 2 yields the value 3002. This is normally not the case with pointer arithmetic. When an integer is added to or subtracted from a pointer, the pointer is not incremented or decremented simply by that integer, but by that integer
times the size of the object to which the pointer refers. The number of bytes depends on
the object's data type. For example, the statement

\[
vPtr += 2;
\]

would produce 3008 (3000 + 2 * 4), assuming an integer is stored in 4 bytes of memory.
In the array v, vPtr would now point to v[2] (Fig. 7.19). If an integer is stored in 2 bytes
of memory, then the preceding calculation would result in memory location 3004 (3000 +
2 * 2). If the array were of a different data type, the preceding statement would increment
the pointer by twice the number of bytes that it takes to store an object of that data type.
When performing pointer arithmetic on a character array, the results will be consistent
with regular arithmetic, because each character is 1 byte long.

If vPtr had been incremented to 3016, which points to v[4], the statement

\[
vPtr -= 4;
\]

would set vPtr back to 3000—the beginning of the array. If a pointer is being incremented
or decremented by one, the increment (++) and decrement (--) operators can be used. Ei-
ther of the statements

\[
++vPtr; \\
vPtr++;
\]

increments the pointer to point to the next location in the array. Either of the statements

\[
--vPtr; \\
vPtr--;
\]

decrements the pointer to point to the previous element of the array.

Pointer variables may be subtracted from one another. For example, if vPtr contains
the location 3000, and v2Ptr contains the address 3008, the statement

\[
x = v2Ptr - vPtr;
\]

would assign to x the number of array elements from vPtr to v2Ptr, in this case 2 (not 8).
Pointer arithmetic is meaningless unless performed on an array. We cannot assume that
two variables of the same type are stored contiguously in memory unless they're adjacent
elements of an array.
A pointer can be assigned to another pointer if both have the same type. The exception to this rule is the **pointer to void** (i.e., `void *`), which is a generic pointer that can represent any pointer type. All pointer types can be assigned a pointer to `void`, and a pointer to `void` can be assigned a pointer of any type. In both cases, a cast operation is not required.

A pointer to `void` cannot be dereferenced. Consider this: The compiler knows that a pointer to `int` refers to 4 bytes of memory on a machine with 4-byte integers, but a pointer to `void` simply contains a memory location for an unknown data type—the precise number of bytes to which the pointer refers is not known by the compiler. The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer.

Pointers can be compared using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the same array. Pointer comparisons compare the addresses stored in the pointers. A comparison of two pointers pointing to elements in the same array could show, for example, that one pointer points to a higher-numbered element of the array than the other pointer does. A common use of pointer comparison is determining whether a pointer is `NULL`.

### 7.9 Relationship between Pointers and Arrays

Arrays and pointers are intimately related in C and often may be used interchangeably. An array name can be thought of as a constant pointer. Pointers can be used to do any operation involving array subscripting.

Assume that integer array `b[5]` and integer pointer variable `bPtr` have been defined. Since the array name (without a subscript) is a pointer to the first element of the array, we can set `bPtr` equal to the address of the first element in array `b` with the statement

```c
bPtr = b;
```
This statement is equivalent to taking the address of the array's first element as follows:

```
Array element b[3] can alternatively be referenced with the pointer expression

*(bPtr + 3)
```

The 3 in the above expression is the offset to the pointer. When the pointer points to the beginning of an array, the offset indicates which element of the array should be referenced, and the offset value is identical to the array subscript. The preceding notation is referred to as **pointer/offset notation**. The parentheses are necessary because the precedence of `*` is higher than the precedence of `+`. Without the parentheses, the above expression would add 3 to the value of the expression `*bPtr` (i.e., 3 would be added to `b[0]`, assuming `bPtr` points to the beginning of the array). Just as the array element can be referenced with a pointer expression, the address

```
&b[3]
```

can be written with the pointer expression

```
bPtr + 3
```

The array itself can be treated as a pointer and used in pointer arithmetic. For example, the expression

```
*(b + 3)
```

also refers to the array element `b[3]`. In general, all subscripted array expressions can be written with a pointer and an offset. In this case, pointer/offset notation was used with the name of the array as a pointer. The preceding statement does not modify the array name in any way; `b` still points to the first element in the array.

Pointers can be subscripted exactly as arrays can. For example, if `bPtr` has the value `b`, the expression

```
bPtr[1]
```

refers to the array element `b[1]`. This is referred to as **pointer/subscript notation**.

Remember that an array name is essentially a constant pointer; it always points to the beginning of the array. Thus, the expression

```
b += 3
```

is invalid because it attempts to modify the value of the array name with pointer arithmetic.

**Common Programming Error 7.10**

*Attempting to modify an array name with pointer arithmetic is a syntax error.*

Figure 7.20 uses the four methods we have discussed for referring to array elements—array subscripting, pointer/offset with the array name as a pointer, **pointer subscripting**, and pointer/offset with a pointer—to print the four elements of the integer array `b`. 
/* Fig. 7.20: fig07_20.cpp
Using subscripting and pointer notations with arrays */

#include <stdio.h>

int main( void )
{
    int b[] = { 10, 20, 30, 40 }; /* initialize array b */
    int *bPtr = b; /* set bPtr to point to array b */
    int i; /* counter */
    int offset; /* counter */

    /* output array b using array subscript notation */
    printf( "Array b printed with:
    Array subscript notation
    ");

    /* loop through array b */
    for ( i = 0; i < 4; i++ ) {
        printf("b[ %d ] = %d\n", i, b[ i ]);
    } /* end for */

    /* output array b using array name and pointer/offset notation */
    printf( "\nPointer/offset notation where
    the pointer is the array name\n"");

    /* loop through array b */
    for ( offset = 0; offset < 4; offset++ ) {
        printf("*( b + %d ) = %d\n", offset, *( b + offset ) );
    } /* end for */

    /* output array b using bPtr and array subscript notation */
    printf( "\nPointer subscript notation\n"");

    /* loop through array b */
    for ( i = 0; i < 4; i++ ) {
        printf("bPtr[ %d ] = %d\n", i, bPtr[ i ] );
    } /* end for */

    /* output array b using bPtr and pointer/offset notation */
    printf( "\nPointer/offset notation\n"");

    /* loop through array b */
    for ( offset = 0; offset < 4; offset++ ) {
        printf("*( bPtr + %d ) = %d\n", offset, *( b + offset ) );
    } /* end for */

    return 0; /* indicates successful termination */
} /* end main */
To further illustrate the interchangeability of arrays and pointers, let’s look at the two string-copying functions—copy1 and copy2—in the program of Fig. 7.21. Both functions copy a string (possibly a character array) into a character array. After a comparison of the function prototypes for copy1 and copy2, the functions appear identical. They accomplish the same task; however, they’re implemented differently.

```c
/* Fig. 7.21: fig07_21.c
   Copying a string using array notation and pointer notation. */
#include <stdio.h>

void copy1( char * const s1, const char * const s2 ); /* prototype */
void copy2( char *s1, const char *s2 ); /* prototype */

int main( void )
{
    char string1[ 10 ]; /* create array string1 */
    char *string2 = "Hello"; /* create a pointer to a string */
    char string3[ 10 ]; /* create array string3 */
    char string4[] = "Good Bye"; /* create a pointer to a string */

    copy1( string1, string2 );
    printf( "string1 = %s\n", string1 );
    copy2( string3, string4 );
    printf( "string3 = %s\n", string3 );
    return 0; /* indicates successful termination */
} /* end main */
```

**Fig. 7.21** | Copying a string using array notation and pointer notation. (Part 1 of 2.)
7.9 Relationship between Pointers and Arrays

Function copy1 uses array subscript notation to copy the string in \( s2 \) to the character array \( s1 \). The function defines counter variable \( i \) as the array subscript. The for statement header (line 29) performs the entire copy operation—its body is the empty statement. The header specifies that \( i \) is initialized to zero and incremented by one on each iteration of the loop. The expression \( s1[i] = s2[i] \) copies one character from \( s2 \) to \( s1 \). When the null character is encountered in \( s2 \), it’s assigned to \( s1 \), and the value of the assignment becomes the value assigned to the left operand (\( s1 \)). The loop terminates because the integer value of the null character is zero (false).

Function copy2 uses pointers and pointer arithmetic to copy the string in \( s2 \) to the character array \( s1 \). Again, the for statement header (line 38) performs the entire copy operation. The header does not include any variable initialization. As in function copy1, the expression \((*s1 = *s2)\) performs the copy operation. Pointer \( s2 \) is dereferenced, and the resulting character is assigned to the dereferenced pointer \(*s1\). After the assignment in the condition, the pointers are incremented to point to the next element of array \( s1 \) and the next character of string \( s2 \), respectively. When the null character is encountered in \( s2 \), it’s assigned to the dereferenced pointer \( s1 \) and the loop terminates.

The first argument to both copy1 and copy2 must be an array large enough to hold the string in the second argument. Otherwise, an error may occur when an attempt is made to write into a memory location that is not part of the array. Also, the second parameter of each function is declared as \texttt{const char *} (a constant string). In both functions, the second argument is copied into the first argument—characters are read from it one at a time, but the characters are never modified. Therefore, the second parameter is declared to point to a constant value so that the principle of least privilege is enforced—neither

```c
void copy1( char * const s1, const char * const s2 )
{
    int i; /* counter */

    /* loop through strings */
    for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ ) {
        ; /* do nothing in body */
    } /* end for */
} /* end function copy1 */

void copy2( char *s1, const char *s2 )
{
    /* loop through strings */
    for ( ; (*s1 = *s2) != '\0'; s1++, s2++ ) {
        ; /* do nothing in body */
    } /* end for */
} /* end function copy2 */
```

```c
string1 = Hello
string3 = Good Bye
```

Fig. 7.21 | Copying a string using array notation and pointer notation. (Part 2 of 2.)
function requires the capability of modifying the second argument, so neither function is provided with that capability.

### 7.10 Arrays of Pointers

Arrays may contain pointers. A common use of an array of pointers is to form an array of strings, referred to simply as a string array. Each entry in the array is a string, but in C a string is essentially a pointer to its first character. So each entry in an array of strings is actually a pointer to the first character of a string. Consider the definition of string array suit, which might be useful in representing a deck of cards.

```c
```

The suit[4] portion of the definition indicates an array of 4 elements. The char * portion of the declaration indicates that each element of array suit is of type “pointer to char.” Qualifier const indicates that the strings pointed to by each element pointer will not be modified. The four values to be placed in the array are "Hearts", "Diamonds", "Clubs" and "Spades". Each is stored in memory as a null-terminated character string that is one character longer than the number of characters between quotes. The four strings are 7, 9, 6 and 7 characters long, respectively. Although it appears as though these strings are being placed in the suit array, only pointers are actually stored in the array (Fig. 7.22). Each pointer points to the first character of its corresponding string. Thus, even though the suit array is fixed in size, it provides access to character strings of any length. This flexibility is one example of C’s powerful data-structuring capabilities.

The suits could have been placed in a two-dimensional array, in which each row would represent a suit and each column would represent a letter from a suit name. Such a data structure would have to have a fixed number of columns per row, and that number would have to be as large as the largest string. Therefore, considerable memory could be wasted when a large number of strings were being stored with most strings shorter than the longest string. We use string arrays to represent a deck of cards in the next section.

### 7.11 Case Study: Card Shuffling and Dealing Simulation

In this section, we use random number generation to develop a card shuffling and dealing simulation program. This program can then be used to implement programs that play specific card games. To reveal some subtle performance problems, we have intentionally used suboptimal shuffling and dealing algorithms. In this chapter’s exercises and in Chapter 10, we develop more efficient algorithms.
Using the top-down, stepwise refinement approach, we develop a program that will shuffle a deck of 52 playing cards and then deal each of the 52 cards. The top-down approach is particularly useful in attacking larger, more complex problems than we have seen in the early chapters.

We use 4-by-13 double-subscripted array \texttt{deck} to represent the deck of playing cards (Fig. 7.23). The rows correspond to the suits—row 0 corresponds to hearts, row 1 to diamonds, row 2 to clubs and row 3 to spades. The columns correspond to the face values of the cards—columns 0 through 9 correspond to ace through ten respectively, and columns 10 through 12 correspond to jack, queen and king. We shall load string array \texttt{suit} with character strings representing the four suits, and string array \texttt{face} with character strings representing the thirteen face values.

This simulated deck of cards may be shuffled as follows. First the array \texttt{deck} is cleared to zeros. Then, a row (0–3) and a column (0–12) are each chosen at random. The number 1 is inserted in array element \texttt{deck[row][column]} to indicate that this card is going to be the first one dealt from the shuffled deck. This process continues with the numbers 2, 3, …, 52 being randomly inserted in the \texttt{deck} array to indicate which cards are to be placed second, third, …, and fifty-second in the shuffled deck. As the \texttt{deck} array begins to fill with card numbers, it’s possible that a card will be selected twice—i.e., \texttt{deck[row][column]} will be nonzero when it’s selected. This selection is simply ignored and other rows and columns are repeatedly chosen at random until an unselected card is found. Eventually, the numbers 1 through 52 will occupy the 52 slots of the \texttt{deck} array. At this point, the deck of cards is fully shuffled.

This shuffling algorithm could execute indefinitely if cards that have already been shuffled are repeatedly selected at random. This phenomenon is known as \textit{indefinite postponement}. In the exercises, we discuss a better shuffling algorithm that eliminates the possibility of indefinite postponement.

\textbf{Performance Tip 7.3} \\
Sometimes an algorithm that emerges in a “natural” way can contain subtle performance problems, such as indefinite postponement. Seek algorithms that avoid indefinite postponement.
To deal the first card, we search the array for `deck[row][column]` equal to 1. This is accomplished with a nested for statement that varies `row` from 0 to 3 and `column` from 0 to 12. What card does that element of the array correspond to? The suit array has been preloaded with the four suits, so to get the suit, we print the character string `suit[row]`. Similarly, to get the face value of the card, we print the character string `face[column]`. We also print the character string " of ". Printing this information in the proper order enables us to print each card in the form "King of Clubs", "Ace of Diamonds" and so on.

Let’s proceed with the top-down, stepwise refinement process. The top is simply

**Shuffle and deal 52 cards**

Our first refinement yields:

- Initialize the suit array
- Initialize the face array
- Initialize the deck array
- Shuffle the deck
- Deal 52 cards

“Shuffle the deck” may be expanded as follows:

- For each of the 52 cards
  - Place card number in randomly selected unoccupied slot of deck

“Deal 52 cards” may be expanded as follows:

- For each of the 52 cards
  - Find card number in deck array and print face and suit of card

Incorporating these expansions yields our complete second refinement:

- Initialize the suit array
- Initialize the face array
- Initialize the deck array
- For each of the 52 cards
  - Place card number in randomly selected unoccupied slot of deck
- For each of the 52 cards
  - Find card number in deck array and print face and suit of card

“Place card number in randomly selected unoccupied slot of deck” may be expanded as:

- Choose slot of deck randomly
- While chosen slot of deck has been previously chosen
  - Choose slot of deck randomly
- Place card number in chosen slot of deck

“Find card number in deck array and print face and suit of card” may be expanded as:

- For each slot of the deck array
  - If slot contains card number
    - Print the face and suit of the card
Incorporating these expansions yields our third refinement:

Initial the suit array
Initial the face array
Initial the deck array

For each of the 52 cards
  Choose slot of deck randomly
  While slot of deck has been previously chosen
    Choose slot of deck randomly
  Place card number in chosen slot of deck

For each of the 52 cards
  For each slot of deck array
    If slot contains desired card number
      Print the face and suit of the card
This completes the refinement process. This program is more efficient if the shuffle and deal portions of the algorithm are combined so that each card is dealt as it’s placed in the deck. We have chosen to program these operations separately because normally cards are dealt after they’re shuffled (not while they’re shuffled).

The card shuffling and dealing program is shown in Fig. 7.24, and a sample execution is shown in Fig. 7.25. Conversion specifier %s is used to print strings of characters in the calls to printf. The corresponding argument in the printf call must be a pointer to char (or a char array). The format specification "%5s of %-8s" (line 73) prints a character string right justified in a field of five characters followed by " of " and a character string left justified in a field of eight characters. The minus sign in %-8s signifies left justification.

```c
/* Fig. 7.24: fig07_24.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* prototypes */
void shuffle( int wDeck[][13] );
void deal( const int wDeck[][13], const char *wFace[],
          const char *wSuit[] );

int main( void )
{
  /* initialize suit array */

  /* initialize face array */
  const char *face[13] =
```

Fig. 7.24 | Card dealing program. (Part 1 of 3.)
```c
/* initialize deck array */
int deck[4][13] = {0};

srand( time(0) ); /* seed random-number generator */
shuffle( deck ); /* shuffle the deck */
deal( deck, face, suit ); /* deal the deck */
return 0; /* indicates successful termination */
} /* end main */

/* shuffle cards in deck */
void shuffle( int wDeck[][13] )
{
    int row; /* row number */
    int column; /* column number */
    int card; /* counter */

    /* for each of the 52 cards, choose slot of deck randomly */
    for ( card = 1; card <= 52; card++ ) {
        /* choose new random location until unoccupied slot found */
        do {
            row = rand() % 4;
            column = rand() % 13;
        } while ( wDeck[ row ][ column ] != 0 ); /* end do...while */

        /* place card number in chosen slot of deck */
        wDeck[ row ][ column ] = card;
    } /* end for */
} /* end function shuffle */

/* deal cards in deck */
void deal( const int wDeck[][13], const char *wFace[],
           const char *wSuit[] )
{
    int card; /* card counter */
    int row; /* row counter */
    int column; /* column counter */

    /* deal each of the 52 cards */
    for ( card = 1; card <= 52; card++ ) {
        /* loop through rows of wDeck */
        for ( row = 0; row <= 3; row++ ) {
            /* loop through columns of wDeck for current row */
            for ( column = 0; column <= 12; column++ ) {
                /* if slot contains current card, display card */
                if ( wDeck[ row ][ column ] == card ) {
                    printf( "%5s of %-8s%c
", wFace[ column ], wSuit[ row ], card % 2 == 0 ? \n' : 't' );
                }
            }
        }
    }
}
```

**Fig. 7.24** | Card dealing program. (Part 2 of 3.)
There’s a weakness in the dealing algorithm. Once a match is found, the two inner for statements continue searching the remaining elements of deck for a match. We correct this deficiency in this chapter’s exercises and in a Chapter 10 case study.

### 7.12 Pointers to Functions

A **pointer to a function** contains the address of the function in memory. In Chapter 6, we saw that an array name is really the address in memory of the first element of the array. Similarly, a function name is really the starting address in memory of the code that performs the function’s task. Pointers to functions can be passed to functions, returned from functions, stored in arrays and assigned to other function pointers.

To illustrate the use of pointers to functions, Fig. 7.26 presents a modified version of the bubble sort program in Fig. 7.15. The new version consists of main and functions bubble, swap, ascending and descending. Function bubbleSort receives a pointer to a function—either function ascending or function descending—as an argument, in addi-
ation to an integer array and the size of the array. The program prompts the user to choose whether the array should be sorted in ascending or in descending order. If the user enters 1, a pointer to function ascending is passed to function bubble, causing the array to be sorted into increasing order. If the user enters 2, a pointer to function descending is passed to function bubble, causing the array to be sorted into decreasing order. The output of the program is shown in Fig. 7.27.

```c
/* Fig. 7.26: fig07_26.c
   Multipurpose sorting program using function pointers */
#include <stdio.h>
#define SIZE 10

/* prototypes */
void bubble( int work[], const int size, int (*compare)( int a, int b ) );
int ascending( int a, int b );
int descending( int a, int b );

int main( void )
{
    int order; /* 1 for ascending order or 2 for descending order */
    int counter; /* counter */

    /* initialize array a */
    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

    printf( "Enter 1 to sort in ascending order,\n"     "Enter 2 to sort in descending order: " );
    scanf( "%d", &order );

    printf( "\nData items in original order\n" );

    /* output original array */
    for ( counter = 0; counter < SIZE; counter++ ) {
        printf( "%5d", a[ counter ] );
    } /* end for */

    /* sort array in ascending order; pass function ascending as an
       argument to specify ascending sorting order */
    if ( order == 1 ) {
        bubble( a, SIZE, ascending );
        printf( "\nData items in ascending order\n" );
    } /* end if */
    else { /* pass function descending */
        bubble( a, SIZE, descending );
        printf( "\nData items in descending order\n" );
    } /* end else */

    /* output sorted array */
    for ( counter = 0; counter < SIZE; counter++ ) {
        printf( "%5d", a[ counter ] );
    } /* end for */
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 1 of 2.)
printf("\n");
return 0; /* indicates successful termination */
} /* end main */

/* multipurpose bubble sort; parameter compare is a pointer to
the comparison function that determines sorting order */
void bubble( int work[], const int size, int (*compare)( int a, int b ) )
{
    int pass; /* pass counter */
    int count; /* comparison counter */
    void swap( int *element1Ptr, int *element2Ptr ); /* prototype */

    /* loop to control passes */
    for ( pass = 1; pass < size; pass++ ) {
        /* loop to control number of comparisons per pass */
        for ( count = 0; count < size - 1; count++ ) {
            /* if adjacent elements are out of order, swap them */
            if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
                swap( &work[ count ], &work[ count + 1 ] );
            } /* end if */
        } /* end for */
    } /* end for */
} /* end function bubble */

/* swap values at memory locations to which element1Ptr and
element2Ptr point */
void swap( int *element1Ptr, int *element2Ptr )
{
    int hold; /* temporary holding variable */
    hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
} /* end function swap */

/* determine whether elements are out of order for an ascending
order sort */
int ascending( int a, int b )
{
    return b < a; /* swap if b is less than a */
} /* end function ascending */

/* determine whether elements are out of order for a descending
order sort */
int descending( int a, int b )
{
    return b > a; /* swap if b is greater than a */
} /* end function descending */

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 2 of 2.)
The following parameter appears in the function header for `bubble` (line 52)

```
int (*compare)( int a, int b )
```

This tells `bubble` to expect a parameter (compare) that is a pointer to a function that receives two integer parameters and returns an integer result. Parentheses are needed around *compare to group * with compare to indicate that compare is a pointer. If we had not included the parentheses, the declaration would have been

```
int *compare( int a, int b )
```

which declares a function that receives two integers as parameters and returns a pointer to an integer.

The function prototype for `bubble` is shown in line 7. The prototype could have been written as

```
int (*)( int, int );
```

without the function-pointer name and parameter names.

The function passed to `bubble` is called in an if statement (line 66) as follows:

```
if ( (*compare)( work[ count ], work[ count + 1 ] ) )
```

Just as a pointer to a variable is dereferenced to access the value of the variable, a pointer to a function is dereferenced to use the function.

The call to the function could have been made without dereferencing the pointer as in

```
if ( compare( work[ count ], work[ count + 1 ] ) )
```

which uses the pointer directly as the function name. We prefer the first method of calling a function through a pointer because it explicitly illustrates that compare is a pointer to a function that is dereferenced to call the function. The second method of calling a function through a pointer makes it appear as though compare is an actual function. This may be confusing to a user of the program who would like to see the definition of function compare and finds that it’s never defined in the file.
Using Function Pointers to Create a Menu-Driven System

A common use of function pointers is in text-based menu-driven systems. A user is prompted to select an option from a menu (possibly from 1 to 5) by typing the menu item’s number. Each option is serviced by a different function. Pointers to each function are stored in an array of pointers to functions. The user’s choice is used as a subscript in the array, and the pointer in the array is used to call the function.

Figure 7.28 provides a generic example of the mechanics of defining and using an array of pointers to functions. We define three functions—function1, function2 and function3—that each take an integer argument and return nothing. We store pointers to these three functions in array f, which is defined in line 14.

```c
/* Fig. 7.28: fig07_28.c
   Demonstrating an array of pointers to functions */
#include <stdio.h>

/* prototypes */

void function1(int a);
void function2(int b);
void function3(int c);

int main(void)
{
    /* initialize array of 3 pointers to functions that each take an
     int argument and return void */
    void (*f[3])(int) = { function1, function2, function3 };

    int choice; /* variable to hold user's choice */
    printf("Enter a number between 0 and 2, 3 to end: ");
    scanf("%d", &choice);

    /* process user's choice */
    while (choice >= 0 && choice < 3)
    {
        /* invoke function at location choice in array f and pass
         choice as an argument */
        (*f[choice])(choice);
        printf("Enter a number between 0 and 2, 3 to end: ");
        scanf("%d", &choice);
    } /* end while */

    printf("Program execution completed.\n");
    return 0; /* indicates successful termination */
} /* end main */

void function1(int a)
{
    printf("You entered %d so function1 was called\n\n", a);
} /* end function1 */
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 1 of 2.)
The definition is read beginning in the leftmost set of parentheses, "f is an array of 3 pointers to functions that each take an int as an argument and return void." The array is initialized with the names of the three functions. When the user enters a value between 0 and 2, the value is used as the subscript into the array of pointers to functions. In the function call (line 26), f[choice] selects the pointer at location choice in the array. The pointer is dereferenced to call the function, and choice is passed as the argument to the function. Each function prints its argument’s value and its function name to demonstrate that the function is called correctly. In this chapter’s exercises, you’ll develop several text-based, menu-driven systems.

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 2 of 2.)
• The operand of the address operator must be a variable.
• The indirection operator * returns the value of the object to which its operand points.
• The printf conversion specifier %p outputs a memory location as a hexadecimal integer on most
  platforms.

Section 7.4 Passing Arguments to Functions by Reference
• All arguments in C are passed by value.
• C provides the capabilities for simulating call-by-reference using pointers and the indirection op-
  erator. To pass a variable by reference, apply the address operator (&) to the variable’s name.
• When the address of a variable is passed to a function, the indirection operator (*) may be used
  in the function to modify the value at that location in the caller’s memory.
• A function receiving an address as an argument must define a pointer parameter to receive the
  address.
• The compiler does not differentiate between a function that receives a pointer and a function that
  receives a single-subscripted array. A function must “know” when it’s receiving an array vs. a sin-
  gle variable passed by reference.
• When the compiler encounters a function parameter for a single-subscripted array of the form
  int b[], the compiler converts the parameter to the pointer notation int *b.

Section 7.5 Using the const Qualifier with Pointers
• The const qualifier indicates that the value of a particular variable should not be modified.
• If an attempt is made to modify a value that is declared const, the compiler catches it and issues
  either a warning or an error, depending on the particular compiler.
• There are four ways to pass a pointer to a function: a non-constant pointer to non-constant data,
  a constant pointer to non-constant data, a non-constant pointer to constant data, and a constant
  pointer to constant data.
• With a non-constant pointer to non-constant data, the data can be modified through the deref-
  erenced pointer, and the pointer can be modified to point to other data items.
• A non-constant pointer to constant data can be modified to point to any data item of the appro-
  riate type, but the data to which it points cannot be modified.
• A constant pointer to non-constant data always points to the same memory location, and the data
  at that location can be modified through the pointer. This is the default for an array name.
• A constant pointer to constant data always points to the same memory location, and the data at
  that memory location cannot be modified.

Section 7.7 sizeof Operator
• Unary operator sizeof determine the size in bytes of a variable or type at compilation time.
• When applied to the name of an array, sizeof returns the total number of bytes in the array.
• Type size_t is an integral type (unsigned or unsigned long) returned by operator sizeof. Type
  size_t is defined in header <stddef.h>.
• Operator sizeof can be applied to any variable name, type or value.
• The parentheses used with sizeof are required if a type name is supplied as its operand.

Section 7.8 Pointer Expressions and Pointer Arithmetic
• A limited set of arithmetic operations may be performed on pointers. A pointer may be incre-
  mented (++) or decremented (--), an integer may be added to a pointer (+ or +=), an integer may
  be subtracted from a pointer (- or -=) and one pointer may be subtracted from another.
• When an integer is added to or subtracted from a pointer, the pointer is incremented or decre-mented by that integer times the size of the object to which the pointer refers.
• Two pointers to elements of the same array may be subtracted from one another to determine the number of elements between them.
• A pointer can be assigned to another pointer if both have the same type. An exception to this is the pointer of type void * which can represent any pointer type. All pointer types can be assigned a void * pointer, and a void * pointer can be assigned a pointer of any type.
• A void * pointer cannot be dereferenced.
• Pointers can be compared using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the same array. Pointer comparisons compare the addresses stored in the pointers.
• A common use of pointer comparison is determining whether a pointer is NULL.

Section 7.9 Relationship between Pointers and Arrays
• Arrays and pointers are intimately related in C and often may be used interchangeably.
• An array name can be thought of as a constant pointer.
• Pointers can be used to do any operation involving array subscripting.
• When a pointer points to the beginning of an array, adding an offset to the pointer indicates which element of the array should be referenced, and the offset value is identical to the array subscript. This is referred to as pointer/offset notation.
• An array name can be treated as a pointer and used in pointer arithmetic expressions that do not attempt to modify the address of the pointer.
• Pointers can be subscripted exactly as arrays can. This is referred to as pointer/subscript notation.
• A parameter of type const char * typically represents a constant string.

Section 7.10 Arrays of Pointers
• Arrays may contain pointers. A common use of an array of pointers is to form an array of strings. Each entry in the array is a string, but in C a string is essentially a pointer to its first character. So each entry in an array of strings is actually a pointer to the first character of a string.

Section 7.12 Pointers to Functions
• A pointer to a function contains the address of the function in memory. A function name is really the starting address in memory of the code that performs the function’s task.
• Pointers to functions can be passed to functions, returned from functions, stored in arrays and assigned to other function pointers.
• A pointer to a function is dereferenced to call the function. A function pointer can be used directly as the function name when calling the function.
• A common use of function pointers is in text-based, menu-driven systems.

Terminology
address operator (&) 255
array of pointers 280
array of strings 280
call-by-reference 257
call-by-value 257
const qualifier 261
constant pointer to constant data 262
constant pointer to non-constant data 262
dereferencing a pointer 256
dereferencing operator (*) 256
function pointer 289
indefinite postponement 281
indirection 254
indirection operator (*) 256
Self-Review Exercises

7.1 Answer each of the following:
   a) A pointer variable contains as its value the _______ of another variable.
   b) The three values that can be used to initialize a pointer are _______, _______ and _______.
   c) The only integer that can be assigned to a pointer is _______.

7.2 State whether the following are true or false. If the answer is false, explain why.
   a) The address operator (&) can be applied only to constants, to expressions and to variables declared with the storage-class register.
   b) A pointer that is declared to be void can be dereferenced.
   c) Pointers of different types may not be assigned to one another without a cast operation.

7.3 Answer each of the following. Assume that single-precision floating-point numbers are stored in 4 bytes, and that the starting address of the array is at location 1002500 in memory. Each part of the exercise should use the results of previous parts where appropriate.
   a) Define an array of type float called numbers with 10 elements, and initialize the elements to the values 0.0, 1.1, 2.2, ..., 9.9. Assume the symbolic constant SIZE has been defined as 10.
   b) Define a pointer, nPtr, that points to an object of type float.
   c) Print the elements of array numbers using array subscript notation. Use a for statement and assume the integer control variable i has been defined. Print each number with 1 position of precision to the right of the decimal point.
   d) Give two separate statements that assign the starting address of array numbers to the pointer variable nPtr.
   e) Print the elements of array numbers using pointer/offset notation with the pointer nPtr.
   f) Print the elements of array numbers using pointer/offset notation with the array name as the pointer.
   g) Print the elements of array numbers by subscripting pointer nPtr.
   h) Refer to element 4 of array numbers using array subscript notation, pointer/offset notation with the array name as the pointer, pointer subscript notation with nPtr and pointer/offset notation with nPtr.
   i) Assuming that nPtr points to the beginning of array numbers, what address is referenced by nPtr + 8? What value is stored at that location?
   j) Assuming that nPtr points to numbers[5], what address is referenced by nPtr -= 4. What is the value stored at that location?

7.4 For each of the following, write a statement that performs the indicated task. Assume that floating-point variables number1 and number2 are defined and that number1 is initialized to 7.3.
   a) Define the variable fPtr to be a pointer to an object of type float.
   b) Assign the address of variable number1 to pointer variable fPtr.
c) Print the value of the object pointed to by fPtr.
d) Assign the value of the object pointed to by fPtr to variable number2.
e) Print the value of number2.
f) Print the address of number1. Use the %p conversion specifier.
g) Print the address stored in fPtr. Use the %p conversion specifier. Is the value printed the same as the address of number1?

7.5 Do each of the following:
a) Write the function header for a function called exchange that takes two pointers to floating-point numbers x and y as parameters and does not return a value.
b) Write the function prototype for the function in part (a).
c) Write the function header for a function called evaluate that returns an integer and that takes as parameters integer x and a pointer to function poly. Function poly takes an integer parameter and returns an integer.
d) Write the function prototype for the function in part (c).

7.6 Find the error in each of the following program segments. Assume

```c
int *zPtr; /* zPtr will reference array z */
int *aPtr = NULL;
void *sPtr = NULL;
int number, i;
int z[5] = { 1, 2, 3, 4, 5 };  
sPtr = z;

a) ++zptr;
b) /* use pointer to get first value of array; assume zPtr is initialized */
   number = zPtr;
c) /* assign array element 2 (the value 3) to number; 
    assume zPtr is initialized */
   number = *zPtr[2];
d) /* print entire array z; assume zPtr is initialized */
   for ( i = 0; i <= 5; i++ ) {
      printf("%d ", zPtr[i]);
   }
  c) /* assign the value pointed to by sPtr to number */
   number = *sPtr;
f) ++z;
```

Answers to Self-Review Exercises

7.1 a) address. b) 0, NULL, an address. c) 0.

7.2 a) False. The address operator can be applied only to variables. The address operator cannot be applied to variables declared with storage class register.
b) False. A pointer to void cannot be dereferenced, because there is no way to know exactly how many bytes of memory to dereference.
c) False. Pointers of type void can be assigned pointers of other types, and pointers of type void can be assigned to pointers of other types.

7.3 a) `float numbers[ SIZE ] = 
{ 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };`
b) `float *nPtr;
c) for ( i = 0; i < SIZE; i++ ) {
   printf("%.1f ", numbers[i]);
}
d) \texttt{nPtr = numbers;}  
    \texttt{nPtr = &numbers[0];}

c) \texttt{for ( i = 0; i < SIZE; i++ ) {  
    printf("%.1f ", *( nPtr + i ));  
}}

d) \texttt{for ( i = 0; i < SIZE; i++ ) {  
    printf("%.1f ", *( numbers + i ));  
}}

e) \texttt{for ( i = 0; i < SIZE; i++ ) {  
    printf("%.1f ", nPtr[ i ]);  
}}

f) \texttt{numbers[4] * ( numbers + 4 )}
    \texttt{nPtr[4] * ( nPtr + 4 )}

i) The address is \(1002500 + 8 \times 4 = 1002532\). The value is 8.8.

j) The address of \texttt{numbers[5]} is \(1002500 + 5 \times 4 = 1002520\).
The address of \texttt{nPtr -= 4} is \(1002520 - 4 \times 4 = 1002504\). The value at that location is 1.1.

7.4 a) \texttt{float *fPtr;}  
b) \texttt{fPtr = &number1;}  
c) \texttt{printf("The value of *fPtr is %f\n", *fPtr);}  
d) \texttt{number2 = *fPtr;}  
e) \texttt{printf("The value of number2 is %f\n", number2);}  
f) \texttt{printf("The address of number1 is %p\n", &number1);}  
g) \texttt{printf("The address stored in fptr is %p\n", fPtr);}  
    Yes, the value is the same.

7.5 a) \texttt{void exchange(float *x, float *y)}  
b) \texttt{void exchange(float *x, float *y);}  
c) \texttt{int evaluate(int x, int (*poly)(int))}  
d) \texttt{int evaluate(int x, int (*poly)(int));}

7.6 a) Error: \texttt{zPtr} has not been initialized.  
    Correction: Initialize \texttt{zPtr} with \texttt{zPtr = z;} before performing the pointer arithmetic.

b) Error: The pointer is not dereferenced.  
    Correction: Change the statement to \texttt{number = *zPtr;}  
    \texttt{zPtr[2]} is not a pointer and should not be dereferenced.  
    Correction: Change \texttt{zPtr[2]} to \texttt{zPtr[2]}.

c) Error: Referring to an array element outside the array bounds with pointer subscripting.  
    Correction: Change the operator \texttt{<=} in the \texttt{for} condition to \texttt{<}.  
    Error: Dereferencing a void pointer.  
    Correction: To dereference the pointer, it must first be cast to an integer pointer.  
    Change the statement to \texttt{number = *(int*)sPtr;}  
    Error: Trying to modify an array name with pointer arithmetic.  
    Correction: Use a pointer variable instead of the array name to accomplish pointer arithmetic, or subscript the array name to refer to a specific element.

7.7 Answer each of the following:
    a) The \texttt{________} operator returns the location in memory where its operand is stored.
Chapter 7  C Pointers

b) The ______ operator returns the value of the object to which its operand points.
c) To simulate call-by-reference when passing a nonarray variable to a function, it’s necessary to pass the ______ of the variable to the function.

7.8 State whether the following are true or false. If false, explain why.
a) Two pointers that point to different arrays cannot be compared meaningfully.
b) Because the name of an array is a pointer to the first element of the array, array names may be manipulated in precisely the same manner as pointers.

7.9 Answer each of the following. Assume that unsigned integers are stored in 2 bytes and that the starting address of the array is at location 1002500 in memory.
a) Define an array of type unsigned int called values with five elements, and initialize the elements to the even integers from 2 to 10. Assume the symbolic constant SIZE has been defined as 5.
b) Define a pointer vPtr that points to an object of type unsigned int.
c) Print the elements of array values using array subscript notation. Use a for statement and assume integer control variable i has been defined.
d) Give two separate statements that assign the starting address of array values to pointer variable vPtr.
e) Print the elements of array values using pointer/offset notation.
f) Print the elements of array values using pointer/offset notation with the array name as the pointer.
g) Print the elements of array values by subscripting the pointer to the array.
h) Refer to element 5 of array values using array subscript notation, pointer/offset notation with the array name as the pointer, pointer subscript notation, and pointer/offset notation.
i) What address is referenced by vPtr + 3? What value is stored at that location?
j) Assuming vPtr points to values[4], what address is referenced by vPtr += 4. What value is stored at that location?

7.10 For each of the following, write a single statement that performs the indicated task. Assume that long integer variables value1 and value2 have been defined and that value1 has been initialized to 200000.
a) Define the variable lPtr to be a pointer to an object of type long.
b) Assign the address of variable value1 to pointer variable lPtr.
c) Print the value of the object pointed to by lPtr.
d) Assign the value of the object pointed to by lPtr to variable value2.
e) Print the value of value2.
f) Print the address of value1.
g) Print the address stored in lPtr. Is the value printed the same as the address of value1?

7.11 Do each of the following:
a) Write the function header for function zero, which takes a long integer array parameter bigIntegers and does not return a value.
b) Write the function prototype for the function in part a.
c) Write the function header for function add1AndSum, which takes an integer array parameter oneTooSmall and returns an integer.
d) Write the function prototype for the function described in part c.

Note: Exercise 7.12 through Exercise 7.15 are reasonably challenging. Once you have done these problems, you ought to be able to implement most popular card games easily.

7.12 (Card Shuffling and Dealing) Modify the program in Fig. 7.24 so that the card-dealing function deals a five-card poker hand. Then write the following additional functions:
Exercises 297

a) Determine if the hand contains a pair.
b) Determine if the hand contains two pairs.
c) Determine if the hand contains three of a kind (e.g., three jacks).
d) Determine if the hand contains four of a kind (e.g., four aces).
e) Determine if the hand contains a flush (i.e., all five cards of the same suit).
f) Determine if the hand contains a straight (i.e., five cards of consecutive face values).

7.13 (Project: Card Shuffling and Dealing) Use the functions developed in Exercise 7.12 to write a program that deals two five-card poker hands, evaluates each hand, and determines which is the better hand.

7.14 (Project: Card Shuffling and Dealing) Modify the program developed in Exercise 7.13 so that it can simulate the dealer. The dealer’s five-card hand is dealt “face down” so the player cannot see it. The program should then evaluate the dealer’s hand, and based on the quality of the hand, the dealer should draw one, two or three more cards to replace the corresponding number of unnecessary cards in the original hand. The program should then re-evaluate the dealer’s hand. [Caution: This is a difficult problem!]

7.15 (Project: Card Shuffling and Dealing) Modify the program developed in Exercise 7.14 so that it can handle the dealer’s hand automatically, but the player is allowed to decide which cards of the player’s hand to replace. The program should then evaluate both hands and determine who wins. Now use this new program to play 20 games against the computer. Who wins more games, you or the computer? Have one of your friends play 20 games against the computer. Who wins more games? Based on the results of these games, make appropriate modifications to refine your poker playing program (this, too, is a difficult problem). Play 20 more games. Does your modified program play a better game?

7.16 (Card Shuffling and Dealing Modification) In the card shuffling and dealing program of Fig. 7.24, we intentionally used an inefficient shuffling algorithm that introduced the possibility of indefinite postponement. In this problem, you’ll create a high-performance shuffling algorithm that avoids indefinite postponement.

Modify the program of Fig. 7.24 as follows. Begin by initializing the deck array as shown in Fig. 7.29. Modify the shuffle function to loop row-by-row and column-by-column through the array, touching every element once. Each element should be swapped with a randomly selected element of the array. Print the resulting array to determine if the deck is satisfactorily shuffled (as in Fig. 7.30, for example). You may want your program to call the shuffle function several times to ensure a satisfactory shuffle.

Although the approach in this problem improves the shuffling algorithm, the dealing algorithm still requires searching the deck array for card 1, then card 2, then card 3, and so on. Worse yet, even after the dealing algorithm locates and deals the card, the algorithm continues searching through the remainder of the deck. Modify the program of Fig. 7.24 so that once a card is dealt, no

<table>
<thead>
<tr>
<th>Unshuffled deck array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12</td>
</tr>
<tr>
<td>0 1 14 15 16 17 18 19 20 21 22 23 24</td>
</tr>
<tr>
<td>1 2 27 28 29 30 31 32 33 34 35 36 37</td>
</tr>
<tr>
<td>2 3 40 41 42 43 44 45 46 47 48 49 50</td>
</tr>
<tr>
<td>3 4 40 41 42 43 44 45 46 47 48 49 50</td>
</tr>
<tr>
<td>5 6 7 8 9 10 11 12 13 14 15 16 17</td>
</tr>
<tr>
<td>6 7 8 9 10 11 12 13 14 15 16 17 18</td>
</tr>
<tr>
<td>7 8 9 10 11 12 13 14 15 16 17 18 19</td>
</tr>
<tr>
<td>8 9 10 11 12 13 14 15 16 17 18 19 20</td>
</tr>
<tr>
<td>9 10 11 12 13 14 15 16 17 18 19 20 21</td>
</tr>
<tr>
<td>10 11 12 13 14 15 16 17 18 19 20 21 22</td>
</tr>
<tr>
<td>11 12 13 14 15 16 17 18 19 20 21 22 23</td>
</tr>
<tr>
<td>12 13 14 15 16 17 18 19 20 21 22 23 24</td>
</tr>
<tr>
<td>13 14 15 16 17 18 19 20 21 22 23 24 25</td>
</tr>
<tr>
<td>14 15 16 17 18 19 20 21 22 23 24 25 26</td>
</tr>
<tr>
<td>15 16 17 18 19 20 21 22 23 24 25 26 27</td>
</tr>
<tr>
<td>16 17 18 19 20 21 22 23 24 25 26 27 28</td>
</tr>
<tr>
<td>17 18 19 20 21 22 23 24 25 26 27 28 29</td>
</tr>
<tr>
<td>18 19 20 21 22 23 24 25 26 27 28 29 30</td>
</tr>
<tr>
<td>19 20 21 22 23 24 25 26 27 28 29 30 31</td>
</tr>
<tr>
<td>20 21 22 23 24 25 26 27 28 29 30 31 32</td>
</tr>
<tr>
<td>21 22 23 24 25 26 27 28 29 30 31 32 33</td>
</tr>
<tr>
<td>22 23 24 25 26 27 28 29 30 31 32 33 34</td>
</tr>
<tr>
<td>23 24 25 26 27 28 29 30 31 32 33 34 35</td>
</tr>
<tr>
<td>24 25 26 27 28 29 30 31 32 33 34 35 36</td>
</tr>
<tr>
<td>25 26 27 28 29 30 31 32 33 34 35 36 37</td>
</tr>
<tr>
<td>26 27 28 29 30 31 32 33 34 35 36 37 38</td>
</tr>
<tr>
<td>27 28 29 30 31 32 33 34 35 36 37 38 39</td>
</tr>
<tr>
<td>28 29 30 31 32 33 34 35 36 37 38 39 40</td>
</tr>
<tr>
<td>29 30 31 32 33 34 35 36 37 38 39 40 41</td>
</tr>
<tr>
<td>30 31 32 33 34 35 36 37 38 39 40 41 42</td>
</tr>
<tr>
<td>31 32 33 34 35 36 37 38 39 40 41 42 43</td>
</tr>
<tr>
<td>32 33 34 35 36 37 38 39 40 41 42 43 44</td>
</tr>
<tr>
<td>33 34 35 36 37 38 39 40 41 42 43 44 45</td>
</tr>
<tr>
<td>34 35 36 37 38 39 40 41 42 43 44 45 46</td>
</tr>
<tr>
<td>35 36 37 38 39 40 41 42 43 44 45 46 47</td>
</tr>
<tr>
<td>36 37 38 39 40 41 42 43 44 45 46 47 48</td>
</tr>
<tr>
<td>37 38 39 40 41 42 43 44 45 46 47 48 49</td>
</tr>
<tr>
<td>38 39 40 41 42 43 44 45 46 47 48 49 50</td>
</tr>
<tr>
<td>39 40 41 42 43 44 45 46 47 48 49 50 51</td>
</tr>
<tr>
<td>40 41 42 43 44 45 46 47 48 49 50 51 52</td>
</tr>
</tbody>
</table>

Fig. 7.29 | Unshuffled deck array.
further attempts are made to match that card number, and the program immediately proceeds with
dealing the next card. In Chapter 10, we develop a dealing algorithm that requires only one opera-
tion per card.

7.17  (Simulation: The Tortoise and the Hare) In this problem, you’ll recreate one of the truly
great moments in history, namely the classic race of the tortoise and the hare. You’ll use random
number generation to develop a simulation of this memorable event.

Our contenders begin the race at “square 1” of 70 squares. Each square represents a possible
position along the race course. The finish line is at square 70. The first contender to reach or pass
square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side
of a slippery mountain, so occasionally the contenders lose ground.

There is a clock that ticks once per second. With each tick of the clock, your program should
adjust the position of the animals according to the rules of Fig. 7.31.

<table>
<thead>
<tr>
<th>Animal</th>
<th>Move type</th>
<th>Percentage of the time</th>
<th>Actual move</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tortoise</td>
<td>Fast plod</td>
<td>50%</td>
<td>3 squares to the right</td>
</tr>
<tr>
<td></td>
<td>Slip</td>
<td>20%</td>
<td>6 squares to the left</td>
</tr>
<tr>
<td></td>
<td>Slow plod</td>
<td>30%</td>
<td>1 square to the right</td>
</tr>
<tr>
<td>Hare</td>
<td>Sleep</td>
<td>20%</td>
<td>No move at all</td>
</tr>
<tr>
<td></td>
<td>Big hop</td>
<td>20%</td>
<td>9 squares to the right</td>
</tr>
<tr>
<td></td>
<td>Big slip</td>
<td>10%</td>
<td>12 squares to the left</td>
</tr>
<tr>
<td></td>
<td>Small hop</td>
<td>30%</td>
<td>1 square to the right</td>
</tr>
<tr>
<td></td>
<td>Small slip</td>
<td>20%</td>
<td>2 squares to the left</td>
</tr>
</tbody>
</table>

Fig. 7.30  | Sample shuffled deck array.

Fig. 7.31  | Tortoise and hare rules for adjusting positions.

Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70).
Start each animal at position 1 (i.e., the “starting gate”). If an animal slips left before square 1,
move the animal back to square 1.

Generate the percentages in the preceding table by producing a random integer, i, in the range
1 ≤ i ≤ 10. For the tortoise, perform a “fast plod” when 1 ≤ i ≤ 5, a “slip” when 6 ≤ i ≤ 7, or a
“slow plod” when 8 ≤ i ≤ 10. Use a similar technique to move the hare.

Begin the race by printing

```
BANG  !!!!!
AND  THEY'RE  OFF  !!!!!
```

Then, for each tick of the clock (i.e., each repetition of a loop), print a 70-position line showing
the letter T in the position of the tortoise and the letter H in the position of the hare. Occasionally,
the contenders will land on the same square. In this case, the tortoise bites the hare and your program should print **OUCH!!!** beginning at that position. All print positions other than the T, the H, or the **OUCH!!!** (in case of a tie) should be blank.

After each line is printed, test if either animal has reached or passed square 70. If so, then print the winner and terminate the simulation. If the tortoise wins, print **TORTOISE WINS!!! YAY!!!** If the hare wins, print **Hare wins. Yuch.** If both animals win on the same tick of the clock, you may want to favor the turtle (the “underdog”), or you may want to print **It's a tie.** If neither animal wins, perform the loop again to simulate the next tick of the clock. When you're ready to run your program, assemble a group of fans to watch the race. You'll be amazed at how involved your audience gets!

### 7.18 (Card Shuffling and Dealing Modification)
Modify the card shuffling and dealing program of Fig. 7.24 so the shuffling and dealing operations are performed by the same function (shuffleAndDeal). The function should contain one nested looping structure that is similar to function shuffle in Fig. 7.24.

### 7.19 What does this program do?

```c
/* ex07_19.c */
/* What does this program do? */
#include <stdio.h>

void mystery1( char *s1, const char *s2 ); /* prototype */

int main( void )
{
    char string1[ 80 ]; /* create char array */
    char string2[ 80 ]; /* create char array */

    printf( "Enter two strings: ");
    scanf( "%s%s", string1, string2 );
    mystery1( string1, string2 );
    printf("%s", string1 );
    return 0; /* indicates successful termination */
} /* end main */

/* What does this function do? */
void mystery1( char *s1, const char *s2 )
{
    while ( *s1 != '\0' ) {
        s1++;
    } /* end while */

    for ( ; *s1 = *s2; s1++, s2++ ) {
        ; /* empty statement */
    } /* end for */
} /* end function mystery1 */
```

### 7.20 What does this program do?

```c
/* ex07_20.c */
/* what does this program do? */
#include <stdio.h>

int mystery2( const char *s ); /* prototype */

int main( void )
{
    char string[ 80 ]; /* create char array */
```
7.21 Find the error in each of the following program segments. If the error can be corrected, explain how.

a) `int *number; printf("%d\n", *number);`

b) `float *realPtr; long *integerPtr; integerPtr = realPtr;`

c) `int * x, y; x = y;`

d) `char s[] = "this is a character array"; int count; for (; *s != '\0'; s++) { printf("%c ", *s);`

e) `short *numPtr, result; void *genericPtr = numPtr; result = *genericPtr + 7;`

f) `float x = 19.34; float xPtr = &x; printf("%f\n", xPtr);`

g) `char *s; printf("%s\n", s);`

7.22 (Maze Traversal) The following grid is a double-subscripted array representation of a maze.
The # symbols represent the walls of the maze, and the periods (.) represent squares in the possible paths through the maze.

There is a simple algorithm for walking through a maze that guarantees finding the exit (assuming there is an exit). If there is not an exit, you’ll arrive at the starting location again. Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand from the wall, eventually you’ll arrive at the exit of the maze. There may be a shorter path than the one you have taken, but you’re guaranteed to get out of the maze.

Write recursive function mazeTraverse to walk through the maze. The function should receive as arguments a 12-by-12 character array representing the maze and the starting location of the maze. As mazeTraverse attempts to locate the exit from the maze, it should place the character X in each square in the path. The function should display the maze after each move so the user can watch as the maze is solved.

7.23 (Generating Mazes Randomly) Write a function mazeGenerator that takes as an argument a double-subscripted 12-by-12 character array and randomly produces a maze. The function should also provide the starting and ending locations of the maze. Try your function mazeTraverse from Exercise 7.22 using several randomly generated mazes.

7.24 (Mazes of Any Size) Generalize functions mazeTraverse and mazeGenerator of Exercise 7.22 and Exercise 7.23 to process mazes of any width and height.

7.25 (Arrays of Pointers to Functions) Rewrite the program of Fig. 6.22 to use a menu-driven interface. The program should offer the user four options as follows:

Enter a choice:
- 0  Print the array of grades
- 1  Find the minimum grade
- 2  Find the maximum grade
- 3  Print the average on all tests for each student
- 4  End program

One restriction on using arrays of pointers to functions is that all the pointers must have the same type. The pointers must be to functions of the same return type that receive arguments of the same type. For this reason, the functions in Fig. 6.22 must be modified so that they each return the same type and take the same parameters. Modify functions minimum and maximum to print the minimum or maximum value and return nothing. For option 3, modify function average of Fig. 6.22 to output the average for each student (not a specific student). Function average should return nothing and take the same parameters as printArray, minimum and maximum. Store the pointers to the four functions in array processGrades and use the choice made by the user as the subscript into the array for calling each function.

7.26 What does this program do?
Special Section: Building Your Own Computer

In the next several problems, we take a temporary diversion away from the world of high-level language programming. We “peel open” a computer and look at its internal structure. We introduce machine-language programming and write several machine-language programs. To make this an especially valuable experience, we then build a computer (through the technique of software-based simulation) on which you can execute your machine-language programs!

7.27 (Machine-Language Programming) Let’s create a computer we’ll call the Simpletron. As its name implies, it’s a simple machine, but as we’ll soon see, a powerful one as well. The Simpletron runs programs written in the only language it directly understands—that is, Simpletron Machine Language, or SML for short.

The Simpletron contains an accumulator—a “special register” in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All information in the Simpletron is handled in terms of words. A word is a signed four-digit decimal number such as +3364, -1293, +0007, -0001 and so on. The Simpletron is equipped with a 100-word memory, and these words are referenced by their location numbers 00, 01, …, 99.

Before running an SML program, we must load or place the program into memory. The first instruction (or statement) of every SML program is always placed in location 00.

Each instruction written in SML occupies one word of the Simpletron’s memory (and hence instructions are signed four-digit decimal numbers). We assume that the sign of an SML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the Simpletron’s memory may contain either an instruction, a data value used by a program or an unused (and hence undefined) area of memory. The first two digits of each SML instruction are the operation code, which specifies the operation to be performed. SML operation codes are summarized in Fig. 7.32.

<table>
<thead>
<tr>
<th>Operation code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input/output operations:</td>
<td></td>
</tr>
<tr>
<td>#define READ 10</td>
<td>Read a word from the terminal into a specific location in memory.</td>
</tr>
<tr>
<td>#define WRITE 11</td>
<td>Write a word from a specific location in memory to the terminal.</td>
</tr>
</tbody>
</table>

Fig. 7.32 | Simpletron Machine Language (SML) operation codes. (Part 1 of 2.)
The last two digits of an SML instruction are the operand, which is the address of the memory location containing the word to which the operation applies. Now let’s consider several simple SML programs. The following SML program reads two numbers from the keyboard, and computes and prints their sum.

**Operation code** | **Meaning**
--- | ---
**Load/store operations:**
#define LOAD 20  | Load a word from a specific location in memory into the accumulator.
#define STORE 21  | Store a word from the accumulator into a specific location in memory.
**Arithmetic operations:**
#define ADD 30  | Add a word from a specific location in memory to the word in the accumulator (leave result in accumulator).
#define SUBTRACT 31  | Subtract a word from a specific location in memory from the word in the accumulator (leave result in accumulator).
#define DIVIDE 32  | Divide a word from a specific location in memory into the word in the accumulator (leave result in accumulator).
#define MULTIPLY 33  | Multiply a word from a specific location in memory by the word in the accumulator (leave result in accumulator).
**Transfer of control operations:**
#define BRANCH 40  | Branch to a specific location in memory.
#define BRANCHNEG 41  | Branch to a specific location in memory if the accumulator is negative.
#define BRANCHZERO 42  | Branch to a specific location in memory if the accumulator is zero.
#define HALT 43  | Halt—i.e., the program has completed its task.

**Fig. 7.32** | Simpletron Machine Language (SML) operation codes. (Part 2 of 2.)

The last two digits of an SML instruction are the operand, which is the address of the memory location containing the word to which the operation applies. Now let’s consider several simple SML programs. The following SML program reads two numbers from the keyboard, and computes and prints their sum.

<table>
<thead>
<tr>
<th>Location</th>
<th>Number</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>+1007</td>
<td>(Read A)</td>
</tr>
<tr>
<td>01</td>
<td>+1008</td>
<td>(Read B)</td>
</tr>
<tr>
<td>02</td>
<td>+2007</td>
<td>(Load A)</td>
</tr>
<tr>
<td>03</td>
<td>+3008</td>
<td>(Add B)</td>
</tr>
<tr>
<td>04</td>
<td>+2109</td>
<td>(Store C)</td>
</tr>
<tr>
<td>05</td>
<td>+1109</td>
<td>(Write C)</td>
</tr>
<tr>
<td>06</td>
<td>+4300</td>
<td>(Halt)</td>
</tr>
<tr>
<td>07</td>
<td>+0000</td>
<td>(Variable A)</td>
</tr>
<tr>
<td>08</td>
<td>+0000</td>
<td>(Variable B)</td>
</tr>
<tr>
<td>09</td>
<td>+0000</td>
<td>(Result C)</td>
</tr>
</tbody>
</table>
The instruction +1007 reads the first number from the keyboard and places it into location 07 (which has been initialized to zero). Then +1008 reads the next number into location 08. The load instruction, +2007, puts the first number into the accumulator, and the add instruction, +3008, adds the second number to the number in the accumulator. All SML arithmetic instructions leave their results in the accumulator. The store instruction, +2109, places the result back into memory location 09, from which the write instruction, +1109, takes the number and prints it (as a signed four-digit decimal number). The halt instruction, +4300, terminates execution.

The following SML program reads two numbers from the keyboard, and determines and prints the larger value. Note the use of the instruction +4107 as a conditional transfer of control, much the same as C’s if statement.

Now write SML programs to accomplish each of the following tasks.

a) Use a sentinel-controlled loop to read 10 positive integers and compute and print their sum.

b) Use a counter-controlled loop to read seven numbers, some positive and some negative, and compute and print their average.

c) Read a series of numbers and determine and print the largest number. The first number read indicates how many numbers should be processed.

7.28 (A Computer Simulator) It may at first seem outrageous, but in this problem you’re going to build your own computer. No, you won’t be soldering components together. Rather, you’ll use the powerful technique of software-based simulation to create a software model of the Simpletron. You’ll not be disappointed. Your Simpletron simulator will turn the computer you’re using into a Simpletron, and you’ll actually be able to run, test and debug the SML programs you wrote in Exercise 7.27.

When you run your Simpletron simulator, it should begin by printing:

```plaintext
*** Welcome to Simpletron! ***
*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** location number and a question mark (?). ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***
*** your program. ***
```
Simulate the memory of the Simpletron with a single-subscripted array `memory` that has 100 elements. Now assume that the simulator is running, and let’s examine the dialog as we enter the program of Example 2 of Exercise 7.27:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
*** Program loading completed ***
*** Program execution begins ***
```

The SML program has now been placed (or loaded) into the array `memory`. Now the Simpletron executes the SML program. It begins with the instruction in location 00 and, continues sequentially, unless directed to some other part of the program by a transfer of control.

Use the variable `accumulator` to represent the accumulator register. Use the variable `instructionCounter` to keep track of the location in memory that contains the instruction being performed. Use the variable `operationCode` to indicate the operation currently being performed—i.e., the left two digits of the instruction word. Use the variable `operand` to indicate the memory location on which the current instruction operates. Thus, `operand` is the rightmost two digits of the instruction currently being performed. Do not execute instructions directly from memory. Rather, transfer the next instruction to be performed from memory to a variable called `instructionRegister`. Then “pick off” the left two digits and place them in the variable `operationCode`, and “pick off” the right two digits and place them in `operand`.

When Simpletron begins execution, the special registers are initialized as follows:

```
accumulator +0000
instructionCounter 00
instructionRegister +0000
operationCode 00
operand 00
```

Now let’s “walk through” the execution of the first SML instruction, +1009 in memory location 00. This is called an instruction execution cycle.

The `instructionCounter` tells us the location of the next instruction to be performed. We fetch the contents of that location from `memory` by using the C statement

```
instructionRegister = memory[instructionCounter];
```

The operation code and the operand are extracted from the instruction register by the statements

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Now the Simpletron must determine that the operation code is actually a `read` (versus a `write`, a `load`, and so on). A switch differentiates among the twelve operations of SML.

In the `switch` statement, the behavior of various SML instructions is simulated as follows (we leave the others to the reader):

```
read: scanf("%d", &memory[operand]);
load: accumulator = memory[operand];
add: accumulator += memory[operand];
```
Various branch instructions: We’ll discuss these shortly.

halt: This instruction prints the message

*** Simpletron execution terminated ***

then prints the name and contents of each register as well as the complete contents of memory. Such a printout is often called a computer dump. To help you program your dump function, a sample dump format is shown in Fig. 7.33. A dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated.

---

**Fig. 7.33 | Sample Simpletron dump format.**

Let’s proceed with the execution of our program’s first instruction, namely the +1009 in location 00. As we have indicated, the switch statement simulates this by performing the C statement

```c
scanf("%d", &memory[operand]);
```

A question mark (?) should be displayed on the screen before the `scanf` is executed to prompt the user for input. The Simpletron waits for the user to type a value and then press the Return key. The value is then read into location 09.

At this point, simulation of the first instruction is completed. All that remains is to prepare the Simpletron to execute the next instruction. Since the instruction just performed was not a transfer of control, we need merely increment the instruction counter register as follows:

```c
++instructionCounter;
```

This completes the simulated execution of the first instruction. The entire process (i.e., the instruction execution cycle) begins anew with the fetch of the next instruction to be executed.

Now let’s consider how the branching instructions—the transfers of control—are simulated. All we need to do is adjust the value in the instruction counter appropriately. Therefore, the unconditional branch instruction (40) is simulated within the switch as

```c
instructionCounter = operand;
```

The conditional “branch if accumulator is zero” instruction is simulated as

```c
if ( accumulator == 0 ) {
   instructionCounter = operand;
}
```

At this point, you should implement your Simpletron simulator and run the SML programs you wrote in Exercise 7.27. You may embellish SML with additional features and provide for these in your simulator.
Your simulator should check for various types of errors. During the program loading phase, for example, each number the user types into the Simpletron’s memory must be in the range -9999 to 9999. Your simulator should use a while loop to test that each number entered is in this range, and, if not, keep prompting the user to reenter the number until the user enters a correct number.

During the execution phase, your simulator should check for various serious errors, such as attempts to divide by zero, attempts to execute invalid operation codes and accumulator overflows (i.e., arithmetic operations resulting in values larger than +9999 or smaller than -9999). Such serious errors are called fatal errors. When a fatal error is detected, your simulator should print an error message such as:

*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***

and should print a full computer dump in the format we have discussed previously. This will help the user locate the error in the program.

7.29 (Modifications to the Simpletron Simulator) In Exercise 7.28, you wrote a software simulation of a computer that executes programs written in Simpletron Machine Language (SML). In this exercise, we propose several modifications and enhancements to the Simpletron Simulator. In Exercises 12.26 and 12.27, we propose building a compiler that converts programs written in a high-level programming language (a variation of BASIC) to Simpletron Machine Language. Some of the following modifications and enhancements may be required to execute the programs produced by the compiler.

a) Extend the Simpletron Simulator’s memory to contain 1000 memory locations to enable the Simpletron to handle larger programs.

b) Allow the simulator to perform remainder calculations. This requires an additional Simpletron Machine Language instruction.

c) Allow the simulator to perform exponentiation calculations. This requires an additional Simpletron Machine Language instruction.

d) Modify the simulator to use hexadecimal values rather than integer values to represent Simpletron Machine Language instructions.

e) Modify the simulator to allow output of a newline. This requires an additional Simpletron Machine Language instruction.

f) Modify the simulator to process floating-point values in addition to integer values.

g) Modify the simulator to handle string input. [Hint: Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII decimal equivalent of a character. Add a machine-language instruction that will input a string and store the string beginning at a specific Simpletron memory location. The first half of the word at that location will be a count of the number of characters in the string (i.e., the length of the string). Each succeeding half word contains one ASCII character expressed as two decimal digits. The machine-language instruction converts each character into its ASCII equivalent and assigns it to a half word.]

h) Modify the simulator to handle output of strings stored in the format of part (g). [Hint: Add a machine-language instruction that prints a string beginning at a specified Simpletron memory location. The first half of the word at that location is the length of the string in characters. Each succeeding half word contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and prints the string by translating each two-digit number into its equivalent character.]

Array of Function Pointer Exercises

7.30 (Calculating Circle Circumference, Circle Area or Sphere Volume Using Function Pointers) Using the techniques you learned in Fig. 7.28, create a text-based, menu-driven program that allows
Chapter 7  C Pointers

the user to choose whether to calculate the circumference of a circle, the area of a circle or the volume of a sphere. The program should then input a radius from the user, perform the appropriate calculation and display the result. Use an array of function pointers in which each pointer represents a function that returns `void` and receives a `double` parameter. The corresponding functions should each display messages indicating which calculation was performed, the value of the radius and the result of the calculation.

7.31  (Calculator Using Function Pointers) Using the techniques you learned in Fig. 7.28, create a text-based, menu-driven program that allows the user to choose whether to add, subtract, multiply or divide two numbers. The program should then input two `double` values from the user, perform the appropriate calculation and display the result. Use an array of function pointers in which each pointer represents a function that returns `void` and receives two `double` parameters. The corresponding functions should each display messages indicating which calculation was performed, the values of the parameters and the result of the calculation.

Making a Difference

7.32  (Polling) The Internet and the web are enabling more people to network, join a cause, voice opinions, and so on. The U.S. presidential candidates in 2008 used the Internet intensively to get out their messages and raise money for their campaigns. In this exercise, you’ll write a simple polling program that allows users to rate five social-consciousness issues from 1 (least important) to 10 (most important). Pick five causes that are important to you (e.g., political issues, global environmental issues). Use a one-dimensional array `topics` (of type `char` *) to store the five causes. To summarize the survey responses, use a 5-row, 10-column two-dimensional array `responses` (of type `int`), each row corresponding to an element in the `topics` array. When the program runs, it should ask the user to rate each issue. Have your friends and family respond to the survey. Then have the program display a summary of the results, including:

a) A tabular report with the five topics down the left side and the 10 ratings across the top, listing in each column the number of ratings received for each topic.

b) To the right of each row, show the average of the ratings for that issue.

c) Which issue received the highest point total? Display both the issue and the point total.

d) Which issue received the lowest point total? Display both the issue and the point total.

7.33  (CarbonFootprint Calculator: Arrays of Function Pointers) Using arrays of function pointers, as you learned in this chapter, you can specify a set of functions that are called with the same types of arguments and return the same type of data. Governments and companies worldwide are becoming increasingly concerned with carbon footprints (annual releases of carbon dioxide into the atmosphere) from buildings burning various types of fuels for heat, vehicles burning fuels for power, and the like. Many scientists blame these greenhouse gases for the phenomenon called global warming. Create three functions that help calculate the carbon footprint of a building, a car and a bicycle, respectively. Each function should input appropriate data from the user then calculate and display the carbon footprint. (Check out a few websites that explain how to calculate carbon footprints.) Each function should receive no parameters and return `void`. Write a program that prompts the user to enter the type of carbon footprint to calculate, then calls the corresponding function in the array of function pointers. For each type of carbon footprint, display some identifying information and the object’s carbon footprint.
C Characters and Strings

Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences.
—William Strunk, Jr.

I have made this letter longer than usual, because I lack the time to make it short.
—Blaise Pascal

The difference between the almost-right word and the right word is really a large matter—it’s the difference between the lightning bug and the lightning.
—Mark Twain

Objectives
In this chapter, you’ll learn:

- To use the functions of the character-handling library (<ctype.h>).
- To use the string-conversion functions of the general utilities library (<stdlib.h>).
- To use the string and character input/output functions of the standard input/output library (<stdio.h>).
- To use the string-processing functions of the string handling library (<string.h>).
- The power of function libraries for achieving software reusability.
8.1 Introduction

In this chapter, we introduce the C Standard Library functions that facilitate string and character processing. The functions enable programs to process characters, strings, lines of text and blocks of memory.

The chapter discusses the techniques used to develop editors, word processors, page layout software, computerized typesetting systems and other kinds of text-processing software. The text manipulations performed by formatted input/output functions like `printf` and `scanf` can be implemented using the functions discussed in this chapter.

8.2 Fundamentals of Strings and Characters

Characters are the fundamental building blocks of source programs. Every program is composed of a sequence of characters that—when grouped together meaningfully—is interpreted by the computer as a series of instructions used to accomplish a task. A program may contain character constants. A character constant is an `int` value represented as a character in single quotes. The value of a character constant is the integer value of the character in the machine’s character set. For example, ‘z’ represents the integer value of z, and ‘\n’ the integer value of newline (122 and 10 in ASCII, respectively).

A string is a series of characters treated as a single unit. A string may include letters, digits and various special characters such as +, -, *, / and $. String literals, or string constants, in C are written in double quotation marks as follows:

“John Q. Doe” (a name)
“99999 Main Street” (a street address)
“Waltham, Massachusetts” (a city and state)
“(201) 555-1212” (a telephone number)

A string in C is an array of characters ending in the null character (‘\0’). A string is accessed via a pointer to the first character in the string. The value of a string is the address

---

1. Pointers and pointer-based entities such as arrays and strings, when misused intentionally or accidentally, can lead to errors and security breaches. See our Secure C Programming Resource Center (www.deitel.com/SecureC/) for articles, books, white papers and forums on this important topic.
of its first character. Thus, in C, it is appropriate to say that a string is a pointer—in fact, a pointer to the string’s first character. In this sense, strings are like arrays, because an array is also a pointer to its first element.

A character array or a variable of type char * can be initialized with a string in a definition. The definitions

```c
char color[] = "blue";
const char *colorPtr = "blue";
```

each initialize a variable to the string "blue". The first definition creates a 5-element array color containing the characters 'b', 'l', 'u', 'e' and '\0'. The second definition creates pointer variable colorPtr that points to the string "blue" somewhere in memory.

The preceding array definition could also have been written

```c
char color[] = { 'b', 'l', 'u', 'e', '\0' };
```

When defining a character array to contain a string, the array must be large enough to store the string and its terminating null character. The preceding definition automatically determines the size of the array based on the number of initializers in the initializer list.

A string can be stored in an array using scanf. For example, the following statement stores a string in character array word[20]:

```c
scanf( "%s", word );
```

The string entered by the user is stored in word. Variable word is an array, which is, of course, a pointer, so the & is not needed with argument word. Recall from Section 6.4 that function scanf will read characters until a space, tab, newline or end-of-file indicator is encountered. So, it is possible that the user input could exceed 19 characters and that your program might crash! For this reason, use the conversion specifier %19s so that scanf reads...
up to 19 characters and saves the last character for the terminating null character. This prevents scanf from writing characters into memory beyond the end of s. (For reading input lines of arbitrary length, there is a nonstandard—but widely supported—function readline, usually included in stdio.h.) For a character array to be printed as a string, the array must contain a terminating null character.

Common Programming Error 8.3
Processing a single character as a string. A string is a pointer—probably a respectively large integer. However, a character is a small integer (ASCII values range 0–255). On many systems this causes an error, because low memory addresses are reserved for special purposes such as operating-system interrupt handlers—so “access violations” occur.

Common Programming Error 8.4
Passing a character as an argument to a function when a string is expected (and vice versa) is a compilation error.

8.3 Character-Handling Library
The character-handling library (<ctype.h>) includes several functions that perform useful tests and manipulations of character data. Each function receives a character—represented as an int—or EOF as an argument. As we discussed in Chapter 4, characters are often manipulated as integers, because a character in C is usually a 1-byte integer. EOF normally has the value –1, and some hardware architectures do not allow negative values to be stored in char variables, so the character-handling functions manipulate characters as integers. Figure 8.1 summarizes the functions of the character-handling library.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int isdigit( int c );</td>
<td>Returns a true value if c is a digit and 0 (false) otherwise.</td>
</tr>
<tr>
<td>int isalpha( int c );</td>
<td>Returns a true value if c is a letter and 0 otherwise.</td>
</tr>
<tr>
<td>int isalnum( int c );</td>
<td>Returns a true value if c is a digit or a letter and 0 otherwise.</td>
</tr>
<tr>
<td>int isxdigit( int c );</td>
<td>Returns a true value if c is a hexadecimal digit character and 0 otherwise. (See Appendix C, Number Systems, for a detailed explanation of binary numbers, octal numbers, decimal numbers and hexadecimal numbers.)</td>
</tr>
<tr>
<td>int islower( int c );</td>
<td>Returns a true value if c is a lowercase letter and 0 otherwise.</td>
</tr>
<tr>
<td>int isupper( int c );</td>
<td>Returns a true value if c is an uppercase letter and 0 otherwise.</td>
</tr>
<tr>
<td>int tolower( int c );</td>
<td>If c is an uppercase letter, tolower returns c as a lowercase letter. Otherwise, tolower returns the argument unchanged.</td>
</tr>
<tr>
<td>int toupper( int c );</td>
<td>If c is a lowercase letter, toupper returns c as an uppercase letter. Otherwise, toupper returns the argument unchanged.</td>
</tr>
<tr>
<td>int isspace( int c );</td>
<td>Returns a true value if c is a white-space character—newline (‘\n’), space (‘ ’), form feed (‘\f’), carriage return (‘\r’), horizontal tab (‘\t’) or vertical tab (‘\v’)—and 0 otherwise.</td>
</tr>
</tbody>
</table>

Fig. 8.1 | Character-handling library (<ctype.h>) functions. (Part 1 of 2.)
Functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`

Figure 8.2 demonstrates functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. Function `isdigit` determines whether its argument is a digit (0–9). Function `isalpha` determines whether its argument is an uppercase letter (A–Z) or lowercase letter (a–z). Function `isalnum` determines whether its argument is an uppercase letter, a lowercase letter or a digit. Function `isxdigit` determines whether its argument is a hexadecimal digit (A–F, a–f, 0–9).

---

```c
/* Fig. 8.2: fig08_02.c
   Using functions isdigit, isalpha, isalnum, and isxdigit */
#include <stdio.h>
#include <ctype.h>

int main( void )
{
    printf( "%s\n", "According to isdigit: ");
    printf( %s, isdigit( '8' ) ? "8 is a " : "8 is not a ", "digit",
            isdigit( '#' ) ? "# is a " : "# is not a ", "digit" );

    printf( "%s\n", "According to isalpha:",
            isalpha( 'A' ) ? "A is a " : "A is not a ", "letter",
            isalpha( 'b' ) ? "b is a " : "b is not a ", "letter",
            isalpha( '&' ) ? "& is a " : "& is not a ", "letter",
            isalpha( '4' ) ? "4 is a " : "4 is not a ", "letter" );

    printf( "%s\n", "According to isalnum:",
            isalnum( 'A' ) ? "A is a " : "A is not a ", "digit or a letter",
            isalnum( '8' ) ? "8 is a " : "8 is not a ", "digit or a letter",
            isalnum( '#' ) ? "# is a " : "# is not a ", "digit or a letter" );

    printf( "%s\n", "According to isxdigit:",
            isxdigit( 'A' ) ? "A is a " : "A is not a ", "hexadecimal digit",
            isxdigit( '#' ) ? "# is a " : "# is not a ", "hexadecimal digit" );
}
```

---

**Fig. 8.2** | Using `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 1 of 2.)
isxdigit('F') ? "F is a " : "F is not a ",
"hexadecimal digit",
isxdigit('J') ? "J is a " : "J is not a ",
"hexadecimal digit",
isxdigit('7') ? "7 is a " : "7 is not a ",
"hexadecimal digit",
isxdigit('$') ? "$ is a " : "$ is not a ",
"hexadecimal digit",
isxdigit('f') ? "f is a " : "f is not a ",
"hexadecimal digit"

return 0; /* indicates successful termination */
}
/* end main */

According to isdigit:
8 is a digit
# is not a digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit

Fig. 8.2 | Using isdigit, isalpha, isalnum and isxdigit. (Part 2 of 2.)

Figure 8.2 uses the conditional operator (?:) to determine whether the string "is a " or the string "is not a " should be printed in the output for each character tested. For example, the expression

isdigit('8') ? "8 is a " : "8 is not a "

indicates that if '8' is a digit, the string "8 is a " is printed, and if '8' is not a digit (i.e., isdigit returns 0), the string "8 is not a " is printed.

Functions islower, isupper, tolower and toupper
Figure 8.3 demonstrates functions islower, isupper, tolower and toupper. Function islower determines whether its argument is a lowercase letter (a−z). Function isupper determines whether its argument is an uppercase letter (A−Z). Function tolower converts an uppercase letter to a lowercase letter and returns the lowercase letter. If the argument is not an uppercase letter, tolower returns the argument unchanged. Function toupper converts a lowercase letter to an uppercase letter and returns the uppercase letter. If the argument is not a lowercase letter, toupper returns the argument unchanged.
/* Fig. 8.3: fig08_03.c
   Using functions islower, isupper, tolower, toupper */

#include <stdio.h>
#include <ctype.h>

int main(void)
{
    printf( "%s
%s
%s
%s

",
    "According to islower:",
    islower( 'p' ) ? "p is a " : "p is not a ",
    "lowercase letter",
    islower( 'P' ) ? "P is a " : "P is not a ",
    "lowercase letter",
    islower( '5' ) ? "5 is a " : "5 is not a ",
    "lowercase letter",
    islower( '!' ) ? "! is a " : "! is not a ",
    "lowercase letter" );

    printf( "%s
%s
%s
%s

",
    "According to isupper:",
    isupper( 'D' ) ? "D is an " : "D is not an ",
    "uppercase letter",
    isupper( 'd' ) ? "d is an " : "d is not an ",
    "uppercase letter",
    isupper( '8' ) ? "8 is an " : "8 is not an ",
    "uppercase letter",
    isupper( '$' ) ? "$ is an " : "$ is not an ",
    "uppercase letter" );

    printf( "%s
%s
%s
%s

", 
    "u converted to uppercase is ", toupper( 'u' ),
    "7 converted to uppercase is ", toupper( '7' ),
    "$ converted to uppercase is ", toupper( '$' ),
    "L converted to lowercase is ", tolower( 'L' ) );
    return 0; /* indicates successful termination */
} /* end main */

According to islower:
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l
Functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`

Figure 8.4 demonstrates functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`. Function `isspace` determines if a character is one of the following white-space characters: space (" "), form feed ("\f"), newline ("\n"), carriage return ("\r"), horizontal tab ("\t") or vertical tab ("\v"). Function `iscntrl` determines if a character is one of the following control characters: horizontal tab ("\t"), vertical tab ("\v"), form feed ("\f"), alert ("\a"), backspace ("\b"), carriage return ("\r") or newline ("\n"). Function `ispunct` determines if a character is a printing character other than a space, a digit or a letter, such as $, #, , [, ], {, }, ;, : or %. Function `isprint` determines if a character can be displayed on the screen (including the space character). Function `isgraph` is the same as `isprint`, except that the space character is not included.

```c
/* Fig. 8.4: fig08_04.c  
Using functions isspace, iscntrl, ispunct, isprint, isgraph */

#include <stdio.h>
#include <ctype.h>

int main( void )
{
    printf( "According to isspace:",
            "Newline", isspace( '\n' ) ? " is a " : " is not a ",
            "whitespace character", "Horizontal tab",
            isspace( '\t' ) ? " is a " : " is not a ",
            "whitespace character",
            isspace( '%' ) ? "% is a " : "% is not a ",
            "whitespace character " );

    printf( "According to iscntrl:",
            "Newline", iscntrl( '\n' ) ? " is a " : " is not a ",
            "control character", iscntrl( '$' ) ? "$ is a " :
            "$ is not a ", "control character" );

    printf( "According to ispunct:",
            ispunct( ';' ) ? "; is a " : "; is not a ",
            "punctuation character",
            ispunct( 'Y' ) ? "Y is a " : "Y is not a ",
            "punctuation character",
            ispunct( '#' ) ? "# is a " :="# is not a ",
            "punctuation character" );

    printf( "According to isprint:",
            isprint( '$' ) ? "$ is a " : "$ is not a ",
            "printing character",
            "Alert", isprint( '\a' ) ? " is a " : " is not a ",
            "printing character" );

    printf( "According to isgraph:",
            isgraph( 'Q' ) ? "Q is a " : "Q is not a ",
            "printing character other than a space",
```

Fig. 8.4 | Using isspace, iscntrl, ispunct, isprint and isgraph. (Part 1 of 2.)
### 8.4 String-Conversion Functions

This section presents the string-conversion functions from the general utilities library (`<stdlib.h>`). These functions convert strings of digits to integer and floating-point values. Figure 8.5 summarizes the string-conversion functions. Note the use of `const` to declare variable `nPtr` in the function headers (read from right to left as "nPtr is a pointer to a character constant"); `const` specifies that the argument value will not be modified.

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double atof( const char *nPtr );</code></td>
<td>Converts the string <code>nPtr</code> to double.</td>
</tr>
<tr>
<td><code>int atoi( const char *nPtr );</code></td>
<td>Converts the string <code>nPtr</code> to int.</td>
</tr>
<tr>
<td><code>long atol( const char *nPtr );</code></td>
<td>Converts the string <code>nPtr</code> to long int.</td>
</tr>
<tr>
<td><code>double strtod( const char *nPtr, char **endPtr );</code></td>
<td>Converts the string <code>nPtr</code> to double.</td>
</tr>
<tr>
<td><code>long strtol( const char *nPtr, char **endPtr, int base );</code></td>
<td>Converts the string <code>nPtr</code> to long.</td>
</tr>
<tr>
<td><code>unsigned long strtoul( const char *nPtr, char **endPtr, int base );</code></td>
<td>Converts the string <code>nPtr</code> to unsigned long.</td>
</tr>
</tbody>
</table>

---

```
8.4 String-Conversion Functions

- `"Space", isgraph(' ') ? " is a " : " is not a ",
- "printing character other than a space"
- return 0; /* indicates successful termination */
```

According to `isspace`:
- Newline is a whitespace character
- Horizontal tab is a whitespace character
- % is not a whitespace character

According to `iscntrl`:
- Newline is a control character
- $ is not a control character

According to `ispunct`:
- ; is a punctuation character
- Y is not a punctuation character
- # is a punctuation character

According to `isprint`:
- $ is a printing character
- Alert is not a printing character

According to `isgraph`:
- Q is a printing character other than a space
- Space is not a printing character other than a space

---

**Fig. 8.5** | String-conversion functions of the general utilities library.
Function `atof`

Function `atof` (Fig. 8.6) converts its argument—a string that represents a floating-point number—to a double value. The function returns the double value. If the converted value cannot be represented—for example, if the first character of the string is a letter—the behavior of function `atof` is undefined.

```c
/* Fig. 8.6: fig08_06.c */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    double d; /* variable to hold converted string */
    d = atof( "99.0" );
    printf( "%s%.3f
%s%.3f
", "The string \"99.0\" converted to double is ", d,
            "The converted value divided by 2 is ", d / 2.0 );
    return 0; /* indicates successful termination */
} /* end main */
```

The string "99.0" converted to double is 99.000
The converted value divided by 2 is 49.500

Fig. 8.6 | Using `atof`.

Function `atoi`

Function `atoi` (Fig. 8.7) converts its argument—a string of digits that represents an integer—to an int value. The function returns the int value. If the converted value cannot be represented, the behavior of function `atoi` is undefined.

```c
/* Fig. 8.7: fig08_07.c */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int i; /* variable to hold converted string */
    i = atoi( "2593" );
    printf( "%s%d
%s%d
", "The string \"2593\" converted to int is ", i,
            "The converted value minus 593 is ", i - 593 );
    return 0; /* indicates successful termination */
} /* end main */
```

Fig. 8.7 | Using `atoi`. (Part 1 of 2.)
Function `atol` (Fig. 8.8) converts its argument—a string of digits representing a long integer—to a `long` value. The function returns the `long` value. If the converted value cannot be represented, the behavior of function `atol` is undefined. If `int` and `long` are both stored in 4 bytes, function `atoi` and function `atol` work identically.

Function `strtod` (Fig. 8.9) converts a sequence of characters representing a floating-point value to `double`. The function receives two arguments—a string (`char *`) and a pointer to a string (`char **`). The string contains the character sequence to be converted to `double`. The pointer is assigned the location of the first character after the converted portion of the string. Line 14

\[
\text{d} = \text{strtod(} \text{string, } \text{&stringPtr })\;
\]

indicates that `d` is assigned the `double` value converted from `string`, and `stringPtr` is assigned the location of the first character after the converted value (51.2) in `string`.

Function `strtol`

Function `strtol` (Fig. 8.10) converts to `long` a sequence of characters representing an integer. The function receives three arguments—a string (char *), a pointer to a string and an integer. The string contains the character sequence to be converted. The pointer is assigned the location of the first character after the converted portion of the string. The integer specifies the base of the value being converted. Line 13 indicates that `x` is assigned the `long` value converted from `string`. The second argument, `remainderPtr`, is assigned the remainder of `string` after the conversion. Using `NULL` for the second argument causes the remainder of the string to be ignored. The third argument, 0, indicates that the value to be converted can be in octal (base 8), decimal (base 10) or hexadecimal (base 16) format. The base can be specified as 0 or any value between 2 and 36. See Appendix C, Number Systems, for a detailed explanation of the octal, decimal and hexadecimal number systems. Numeric representations of integers from base 11 to base 36 use the characters A–Z to represent the values 10 to 35. For example, hexadecimal values can consist of the digits 0–9 and the characters A–F. A base-11 integer can consist of the digits 0–9 and the character A. A base-24 integer can consist of the digits 0–9 and the characters A–N. A base-36 integer can consist of the digits 0–9 and the characters A–Z.
Function `strtoul` (Fig. 8.11) converts to unsigned long a sequence of characters representing an unsigned long integer. The function works identically to function `strtol`. The statement

\[
x = \text{strtoul( string, \&remainderPtr, 0 );}
\]

in Fig. 8.11 indicates that \( x \) is assigned the unsigned long value converted from `string`. The second argument, `\&remainderPtr`, is assigned the remainder of `string` after the conversion. The third argument, 0, indicates that the value to be converted can be in octal, decimal or hexadecimal format.

```
/* Fig. 8.11: fig08_11.c */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    const char *string = "-1234567abc"; /* initialize string pointer */
    char *remainderPtr; /* create char pointer */
    long x; /* variable to hold converted sequence */
    x = strtoul( string, &remainderPtr, 0 );
    printf( "The original string is \"%s\"\n\nThe converted value is \"%ld\n\nThe remainder of the original string is \"%s\n\nThe converted value plus 567 is \"%ld\n", string, x, remainderPtr, x + 567 );
    return 0; /* indicates successful termination */
} /* end main */
```

Fig. 8.11 | Using `strtoul`. (Part 1 of 2.)
This section presents several functions from the standard input/output library (\texttt{<stdio.h>}) specifically for manipulating character and string data. Figure 8.12 summarizes the character and string input/output functions of the standard input/output library.

![Fig. 8.12](image)

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{int getchar( void );}</td>
<td>Inputs the next character from the standard input and returns it as an integer.</td>
</tr>
<tr>
<td>\texttt{char *fgets( char *s, int n, \texttt{FILE *stream);}}</td>
<td>Inputs characters from the specified stream into the array \texttt{s} until a newline or end-of-file character is encountered, or until \texttt{n - 1} bytes are read. In this chapter, we specify the stream as \texttt{stdin}—the standard input stream, which is typically used to read characters from the keyboard. A terminating null character is appended to the array. Returns the string that was read into \texttt{s}.</td>
</tr>
<tr>
<td>\texttt{int putchar( int c );}</td>
<td>Prints the character stored in \texttt{c} and returns it as an integer.</td>
</tr>
<tr>
<td>\texttt{int puts( const char *s );}</td>
<td>Prints the string \texttt{s} followed by a newline character. Returns a non-zero integer if successful, or \texttt{EOF} if an error occurs.</td>
</tr>
<tr>
<td>\texttt{int printf( char *s, const char *format, ... );}</td>
<td>Equivalent to \texttt{printf}, except the output is stored in the array \texttt{s} instead of printed on the screen. Returns the number of characters written to \texttt{s}, or \texttt{EOF} if an error occurs.</td>
</tr>
<tr>
<td>\texttt{int sscanf( char *s, const char *format, ... );}</td>
<td>Equivalent to \texttt{scanf}, except the input is read from the array \texttt{s} rather than from the keyboard. Returns the number of items successfully read by the function, or \texttt{EOF} if an error occurs.</td>
</tr>
</tbody>
</table>

**Fig. 8.11** | Using \texttt{strtol}. (Part 2 of 2.)
Functions fgets and putchar

Figure 8.13 uses functions fgets and putchar to read a line of text from the standard input (keyboard) and recursively output the characters of the line in reverse order. Function fgets reads characters from the standard input into its first argument—an array of chars—until a newline or the end-of-file indicator is encountered, or until the maximum number of characters is read. The maximum number of characters is one fewer than the value specified in fgets’s second argument. The third argument specifies the stream from which to read characters—in this case, we use the standard input stream (stdin). A null character (‘\0’) is appended to the array when reading terminates. Function putchar prints its character argument. The program calls recursive function reverse to print the line of text backward. If the first character of the array received by reverse is the null character ‘\0’, reverse returns. Otherwise, reverse is called again with the address of the subarray beginning at element s[1], and character s[0] is output with putchar when the recursive call is completed. The order of the two statements in the else portion of the if statement causes reverse to walk to the terminating null character of the string before a character is printed. As the recursive calls are completed, the characters are output in reverse order.

```c
/* Fig. 8.13: fig08_13.c */
#include <stdio.h>

int main( void )
{
    char sentence[80]; /* create char array */
    printf( "Enter a line of text:\n" );
    fgets( sentence, 80, stdin );
    printf( "\nThe line printed backward is:\n" );
    reverse( sentence );
    return 0; /* indicates successful termination */
} /* end main */

/* recursively outputs characters in string in reverse order */
void reverse( const char * const sPtr )
{
    /* if end of the string */
    if ( sPtr[0] == '\0' ) { /* base case */
        return;
    } /* end if */
    else { /* if not end of the string */
        reverse( &sPtr[1] ); /* recursion step */
        putchar( sPtr[0] ); /* use putchar to display character */
    } /* end else */
} /* end function reverse */
```

Fig. 8.13 | Using fgets and putchar. (Part 1 of 2.)
Functions *getchar* and *puts*

Figure 8.14 uses functions *getchar* and *puts* to read characters from the standard input into character array sentence and print the array of characters as a string. Function *getchar* reads a character from the standard input and returns the character as an integer. Function *puts* takes a string (char *) as an argument and prints the string followed by a newline character. The program stops inputting characters when *getchar* reads the newline character entered by the user to end the line of text. A null character is appended to array sentence (line 19) so that the array may be treated as a string. Then, function *puts* prints the string contained in sentence.

```c
/* Fig. 8.14: fig08_14.c */
#include <stdio.h>

int main(void)
{
    char c; /* variable to hold character input by user */
    char sentence[80]; /* create char array */
    int i = 0; /* initialize counter i */

    /* prompt user to enter line of text */
    puts("Enter a line of text:");

    /* use getchar to read each character */
    while ((c = getchar()) != '\n') {
        sentence[i++] = c;
    } /* end while */

    sentence[i] = '\0'; /* terminate string */

    /* use puts to display sentence */
    puts("The line entered was:");
    puts(sentence);
    return 0; /* indicates successful termination */
} /* end main */
```

Fig. 8.13 | Using fgets and putchar. (Part 2 of 2.)

Fig. 8.14 | Using getchar and puts. (Part 1 of 2.)
Function sprintf
Figure 8.15 uses function \texttt{sprintf} to print formatted data into array \texttt{s}—an array of characters. The function uses the same conversion specifiers as \texttt{printf} (see Chapter 9 for a detailed discussion of formatting). The program inputs an \texttt{int} value and a \texttt{double} value to be formatted and printed to array \texttt{s}. Array \texttt{s} is the first argument of \texttt{sprintf}.

```c
/* Fig. 8.15: fig08_15.c
Using sprintf */
#include <stdio.h>

int main( void )
{
    char s[ 80 ]; /* create char array */
    int x; /* x value to be input */
    double y; /* y value to be input */

    printf( "Enter an integer and a double:
" );
    scanf( "%d%lf", &x, &y );

    sprintf( s, "integer:%6d
double:%8.2f", x, y );

    printf( "%s
%s
" ,
            "The formatted output stored in array s is:", s );
    return 0; /* indicates successful termination */
} /* end main */
```

Enter an integer and a double:
298 87.375
The formatted output stored in array \texttt{s} is:
integer: 298
double: 87.38

Function sscanf
Figure 8.16 uses function \texttt{sscanf} to read formatted data from character array \texttt{s}. The function uses the same conversion specifiers as \texttt{scanf}. The program reads an \texttt{int} and a \texttt{double} from array \texttt{s} and stores the values in \texttt{x} and \texttt{y}, respectively. The values of \texttt{x} and \texttt{y} are printed. Array \texttt{s} is the first argument of \texttt{sscanf}.
Chapter 8  C Characters and Strings

8.6 String-Manipulation Functions of the String-Handling Library

The string-handling library (<string.h>) provides many useful functions for manipulating string data (copying strings and concatenating strings), comparing strings, searching strings for characters and other strings, tokenizing strings (separating strings into logical pieces) and determining the length of strings. This section presents the string-manipulation functions of the string-handling library. The functions are summarized in Fig. 8.17. Every function—except for strncpy—appends the null character to its result.

```
/* Fig. 8.16: fig08_16.c
Using sscanf */

#include <stdio.h>

int main( void )
{
    char s[] = "31298 87.375"; /* initialize array s */
    int x; /* x value to be input */
    double y; /* y value to be input */

    sscanf( s, "%d%lf", &x, &y );
    printf( "%s
%s%6d
%s%8.3f
",
        "The values stored in character array s are:",
        "integer:", x, "double:", y);
    return 0; /* indicates successful termination */
} /* end main */
```

The values stored in character array s are:
integer: 31298
double: 87.375

Fig. 8.16  Using sscanf.

8.6 String-Manipulation Functions of the String-Handling Library

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *strcpy( char *s1, const char *s2 )</td>
<td>Copies string s2 into array s1. The value of s1 is returned.</td>
</tr>
<tr>
<td>char *strncpy( char *s1, const char *s2, size_t n )</td>
<td>Copies at most n characters of string s2 into array s1. The value of s1 is returned.</td>
</tr>
<tr>
<td>char *strcat( char *s1, const char *s2 )</td>
<td>Appends string s2 to array s1. The first character of s2 overwrites the terminating null character of s1. The value of s1 is returned.</td>
</tr>
<tr>
<td>char *strncat( char *s1, const char *s2, size_t n )</td>
<td>Appends at most n characters of string s2 to array s1. The first character of s2 overwrites the terminating null character of s1. The value of s1 is returned.</td>
</tr>
</tbody>
</table>

Fig. 8.17  String-manipulation functions of the string-handling library.
Functions \texttt{strcpy} and \texttt{strncpy} specify a parameter of type \texttt{size_t}, which is a type defined by the C standard as the integral type of the value returned by operator \texttt{sizeof}.

\textbf{Portability Tip 8.2}

Type \texttt{size_t} is a system-dependent synonym for either type \texttt{unsigned long} or \texttt{unsigned int}.

\textbf{Error-Prevention Tip 8.2}

When using functions from the string-handling library, include the \texttt{<string.h>} header.

Function \texttt{strcpy} copies its second argument (a string) into its first argument (a character array that must be large enough to store the string and its terminating null character, which is also copied). Function \texttt{strncpy} is equivalent to \texttt{strcpy}, except that \texttt{strncpy} specifies the number of characters to be copied from the string into the array. Function \texttt{strncpy} does not necessarily copy the terminating null character of its second argument. A terminating null character is written only if the number of characters to be copied is at least one more than the length of the string. For example, if "test" is the second argument, a terminating null character is written only if the third argument to \texttt{strncpy} is at least 5 (four characters in "test" plus a terminating null character). If the third argument is larger than 5, null characters are appended to the array until the total number of characters specified by the third argument are written.

\textbf{Common Programming Error 8.5}

Not appending a terminating null character to the first argument of a \texttt{strncpy} when the third argument is less than or equal to the length of the string in the second argument.

\textit{Functions \texttt{strcpy} and \texttt{strncpy}}

Figure 8.18 uses \texttt{strcpy} to copy the entire string in array \texttt{x} into array \texttt{y} and uses \texttt{strncpy} to copy the first 14 characters of array \texttt{x} into array \texttt{z}. A null character (‘\0’) is appended to array \texttt{z}, because the call to \texttt{strncpy} in the program does not write a terminating null character (the third argument is less than the string length of the second argument).

```c
/* Fig. 8.18: fig08_18.c */
/* Using strcpy and strncpy */
#include <stdio.h>
#include <string.h>

int main( void )
{
    char x[] = "Happy Birthday to You"; /* initialize char array x */
    char y[ 25 ]; /* create char array y */
    char z[ 15 ]; /* create char array z */

    /* copy contents of x into y */
    printf( "%s\n%sn\n%sn\n", "The string in array x is: ", x,
            "The string in array y is: ", strcpy( y, x ) );
}
```

\textbf{Fig. 8.18} | Using \texttt{strcpy} and \texttt{strncpy}. (Part 1 of 2.)
Chapter 8  C Characters and Strings

Functions strcat and strncat

Function strcat appends its second argument (a string) to its first argument (a character array containing a string). The first character of the second argument replaces the null (\'\0\') that terminates the string in the first argument. You must ensure that the array used to store the first string is large enough to store the first string, the second string and the terminating null character copied from the second string. Function strncat appends a specified number of characters from the second string to the first string. A terminating null character is automatically appended to the result. Figure 8.19 demonstrates function strcat and function strncat.

Fig. 8.18  |  Using strcpy and strncpy. (Part 2 of 2.)

Fig. 8.19  |  Using strcat and strncat. (Part 1 of 2.)
8.7 Comparison Functions of the String-Handling Library

This section presents the string-handling library’s string-comparison functions, `strcmp` and `strncmp`. Fig. 8.20 contains their prototypes and a brief description of each function.

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int strcmp(const char *s1, const char *s2);</code></td>
<td>Compares the string <code>s1</code> with the string <code>s2</code>. The function returns 0, less than 0 or greater than 0 if <code>s1</code> is equal to, less than or greater than <code>s2</code>, respectively.</td>
</tr>
<tr>
<td><code>int strncmp(const char *s1, const char *s2, size_t n);</code></td>
<td>Compares up to <code>n</code> characters of the string <code>s1</code> with the string <code>s2</code>. The function returns 0, less than 0 or greater than 0 if <code>s1</code> is equal to, less than or greater than <code>s2</code>, respectively.</td>
</tr>
</tbody>
</table>

Figure 8.21 compares three strings using `strcmp` and `strncmp`. Function `strcmp` compares its first string argument with its second string argument, character by character. The function returns 0 if the strings are equal, a negative value if the first string is less than the second string and a positive value if the first string is greater than the second string. Function `strncmp` is equivalent to `strcmp`, except that `strncmp` compares up to a specified number of characters. Function `strncmp` does not compare characters following a null character in a string. The program prints the integer value returned by each function call.

```c
/* Fig. 8.21: fig08_21.c */
#include <stdio.h>
#include <string.h>

int main(void)
{
    const char *s1 = "Happy New Year"; /* initialize char pointer */
    const char *s2 = "Happy New Year"; /* initialize char pointer */
    const char *s3 = "Happy Holidays"; /* initialize char pointer */
```
To understand just what it means for one string to be “greater than” or “less than” another string, consider the process of alphabetizing a series of last names. The reader would, no doubt, place “Jones” before “Smith,” because the first letter of “Jones” comes before the first letter of “Smith” in the alphabet. But the alphabet is more than just a list of 26 letters—it is an ordered list of characters. Each letter occurs in a specific position within the list. “Z” is more than merely a letter of the alphabet; “Z” is specifically the 26th letter of the alphabet.

How does the computer know that one particular letter comes before another? All characters are represented inside the computer as numeric codes; when the computer compares two strings, it actually compares the numeric codes of the characters in the strings.

Common Programming Error 8.6

Assuming that `strcmp` and `strncmp` return 1 when their arguments are equal is a logic error. Both functions return 0 (strangely, the equivalent of C’s false value) for equality. Therefore, when testing two strings for equality, the result of function `strcmp` or `strncmp` should be compared with 0 to determine if the strings are equal.

Portability Tip 8.3

The internal numeric codes used to represent characters may be different on different computers.

In an effort to standardize character representations, most computer manufacturers have designed their machines to utilize one of two popular coding schemes—ASCII or EBCDIC. ASCII stands for “American Standard Code for Information Interchange,” and
EBCDIC stands for “Extended Binary Coded Decimal Interchange Code.” There are other coding schemes, but these two are the most popular. The Unicode® Standard outlines a specification to produce consistent encoding of the vast majority of the world’s characters and symbols. To learn more about Unicode, visit www.unicode.org.

ASCII, EBCDIC and Unicode are called character sets. String and character manipulations actually involve the manipulation of the appropriate numeric codes and not the characters themselves. This explains the interchangeability of characters and small integers in C. Since it is meaningful to say that one numeric code is greater than, less than or equal to another numeric code, it becomes possible to relate various characters or strings to one another by referring to the character codes. Appendix B lists the ASCII character codes.

8.8 Search Functions of the String-Handling Library

This section presents the functions of the string-handling library used to search strings for characters and other strings. The functions are summarized in Fig. 8.22. The functions `strcspn` and `strspn` return `size_t`.

### Function prototype and description

**char *strchr(const char *s, int c);**
Locates the first occurrence of character c in string s. If c is found, a pointer to c in s is returned. Otherwise, a NULL pointer is returned.

**size_t strcspn(const char *s1, const char *s2);**
Determines and returns the length of the initial segment of string s1 consisting of characters not contained in string s2.

**size_t strspn(const char *s1, const char *s2);**
Determines and returns the length of the initial segment of string s1 consisting only of characters contained in string s2.

**char *strpbrk(const char *s1, const char *s2);**
Locates the first occurrence in string s1 of any character in string s2. If a character from string s2 is found, a pointer to the character in string s1 is returned. Otherwise, a NULL pointer is returned.

**char *strrchr(const char *s, int c);**
Locates the last occurrence of c in string s. If c is found, a pointer to c in string s is returned. Otherwise, a NULL pointer is returned.

**char *strstr(const char *s1, const char *s2);**
Locates the first occurrence in string s1 of string s2. If the string is found, a pointer to the string in s1 is returned. Otherwise, a NULL pointer is returned.

**char *strtok(char *s1, const char *s2);**
A sequence of calls to `strtok` breaks string s1 into “tokens”—logical pieces such as words in a line of text—separated by characters contained in string s2. The first call contains s1 as the first argument, and subsequent calls to continue tokenizing the same string contain NULL as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, NULL is returned.
Function `strchr` searches for the first occurrence of a character in a string. If the character is found, `strchr` returns a pointer to the character in the string; otherwise, `strchr` returns `NULL`. Figure 8.23 uses `strchr` to search for the first occurrences of 'a' and 'z' in the string "This is a test".

```c
/* Fig. 8.23: fig08_23.c */
#include <stdio.h>
#include <string.h>

int main( void )
{
    const char *string = "This is a test"; /* initialize char pointer */
    char character1 = 'a'; /* initialize character1 */
    char character2 = 'z'; /* initialize character2 */

    /* if character1 was found in string */
    if ( strchr( string, character1 ) != NULL ) { /* end if */
        printf( "'%c' was found in "%s".\n",
                character1, string );
    } /* end if */
    else { /* if character1 was not found */
        printf( "'%c' was not found in "%s".\n",
                character1, string );
    } /* end else */

    /* if character2 was found in string */
    if ( strchr( string, character2 ) != NULL ) { /* end if */
        printf( "'%c' was found in "%s".\n",
                character2, string );
    } /* end if */
    else { /* if character2 was not found */
        printf( "'%c' was not found in "%s".\n",
                character2, string );
    } /* end else */

    return 0; /* indicates successful termination */
} /* end main */
```

'a' was found in "This is a test".
'z' was not found in "This is a test".

**Fig. 8.23** Using `strchr`.

Function `strcspn` (Fig. 8.24) determines the length of the initial part of the string in its first argument that does not contain any characters from the string in its second argument. The function returns the length of the segment.
**Function strpbrk**

Function **strpbrk** searches its first string argument for the first occurrence of any character in its second string argument. If a character from the second argument is found, `strpbrk` returns a pointer to the character in the first argument; otherwise, `strpbrk` returns `NULL`. Figure 8.25 shows a program that locates the first occurrence in `string1` of any character from `string2`.

---

```c
/* Fig. 8.24: fig08_24.c */
#include <stdio.h>
#include <string.h>

int main( void )
{
    /* initialize two char pointers */
    const char *string1 = "The value is 3.14159";
    const char *string2 = "1234567890";

    printf( "string1 = ", string1, ", string2 = ", string2,
            "The length of the initial segment of string1 containing no characters from string2 = ",
            strcspn( string1, string2 ) );
    return 0; /* indicates successful termination */
} /* end main */
```

**Fig. 8.24** Using `strcspn`.

```
string1 = The value is 3.14159
string2 = 1234567890
The length of the initial segment of string1 containing no characters from string2 = 13
```

---

```c
/* Fig. 8.25: fig08_25.c */
#include <stdio.h>
#include <string.h>

int main( void )
{
    /* initialize two char pointers */
    const char *string1 = "This is a test"; /* initialize char pointer */
    const char *string2 = "beware"; /* initialize char pointer */

    printf( "Of the characters in ", string2,
            " appears earliest in ", string1 );
    return 0; /* indicates successful termination */
} /* end main */
```

**Fig. 8.25** Using `strpbrk`. (Part 1 of 2.)
Function `strrchr`

Function `strrchr` searches for the last occurrence of the specified character in a string. If the character is found, `strrchr` returns a pointer to the character in the string; otherwise, `strrchr` returns `NULL`. Figure 8.26 shows a program that searches for the last occurrence of the character 'z' in the string "A zoo has many animals including zebras."

```c
/* Fig. 8.26: fig08_26.c
Using strrchr */
#include <stdio.h>
#include <string.h>

int main(void)
{
    /* initialize char pointer */
    const char *string1 = "A zoo has many animals including zebras";

    int c = 'z'; /* character to search for */
    printf("The remainder of string1 beginning with the last occurrence of character '%c' is: "%s"
           , c, strrchr( string1, c ) );
    return 0; /* indicates successful termination */
} /* end main */
```

The remainder of string1 beginning with the last occurrence of character 'z' is: "zebras"

Function `strspn`

Function `strspn` (Fig. 8.27) determines the length of the initial part of the string in its first argument that contains only characters from the string in its second argument. The function returns the length of the segment.

```c
/* Fig. 8.27: fig08_27.c
Using strspn */
#include <stdio.h>
#include <string.h>

/* Fig. 8.27 | Using strspn. (Part 1 of 2.) */
```
Function `strstr`

Function `strstr` searches for the first occurrence of its second string argument in its first string argument. If the second string is found in the first string, a pointer to the location of the string in the first argument is returned. Figure 8.28 uses `strstr` to find the string "def" in the string "abcdefabcdef".

```
/* Fig. 8.28: fig08_28.c Using strstr */
#include <stdio.h>
#include <string.h>

int main( void )
{
    const char *string1 = "abcdefabcdef"; /* string to search */
    const char *string2 = "def"; /* string to search for */
    printf( "%s\n%s\n%s\n", "string1 = ", string1, "string2 = ", string2,
            "The remainder of string1 beginning with the first occurrence of string2 is: ",
            strstr( string1, string2 ) );
    return 0; /* indicates successful termination */
} /* end main */
```

string1 = abcdefabcde
string2 = def

The remainder of string1 beginning with the first occurrence of string2 is: defabcdef
Function `strtok`  
Function `strtok` (Fig. 8.29) is used to break a string into a series of tokens. A token is a sequence of characters separated by delimiters (usually spaces or punctuation marks, but a delimiter can be any character). For example, in a line of text, each word can be considered a token, and the spaces and punctuation separating the words can be considered delimiters.

```c
/* Fig. 8.29: fig08_29.c  
Using strtok */
#include <stdio.h>
#include <string.h>

int main( void )
{
    /* initialize array string */
    char string[] = "This is a sentence with 7 tokens";
    char *tokenPtr; /* create char pointer */

    printf( "%s
%s
%s
", 
            "The string to be tokenized is:", string,
            "The tokens are:" );

    tokenPtr = strtok( string, " "); /* begin tokenizing sentence */

    /* continue tokenizing sentence until tokenPtr becomes NULL */
    while ( tokenPtr != NULL ) {
        printf( "%s
", tokenPtr );
        tokenPtr = strtok( NULL, " "); /* get next token */
    } /* end while */

    return 0; /* indicates successful termination */
} /* end main */
```

The string to be tokenized is:  
This is a sentence with 7 tokens

The tokens are:  
This is a sentence with 7 tokens

Fig. 8.29  |  Using `strtok`.

Multiple calls to `strtok` are required to tokenize a string—i.e., break it into tokens (assuming that the string contains more than one token). The first call to `strtok` contains two arguments: a string to be tokenized, and a string containing characters that separate the tokens. In Fig. 8.29, the statement

```
tokenPtr = strtok( string, " "); /* begin tokenizing sentence */
```
assigns tokenPtr a pointer to the first token in string. The second argument, " ", indicates that tokens are separated by spaces. Function `strtok` searches for the first character in string that is not a delimiting character (space). This begins the first token. The function then finds the next delimiting character in the string and replaces it with a null (\'\0\') character to terminate the current token. Function `strtok` saves a pointer to the next character following the token in string and returns a pointer to the current token.

Subsequent `strtok` calls in line 21 continue tokenizing string. These calls contain `NULL` as their first argument. The `NULL` argument indicates that the call to `strtok` should continue tokenizing from the location in string saved by the last call to `strtok`. If no tokens remain when `strtok` is called, `strtok` returns `NULL`. You can change the delimiter string in each new call to `strtok`. Figure 8.29 uses `strtok` to tokenize the string "This is a sentence with 7 tokens". Each token is printed separately. Function `strtok` modifies the input string by placing \0 at the end of each token; therefore, a copy of the string should be made if the string will be used again in the program after the calls to `strtok`.

### 8.9 Memory Functions of the String-Handling Library

The string-handling library functions presented in this section manipulate, compare and search blocks of memory. The functions treat blocks of memory as character arrays and can manipulate any block of data. Figure 8.30 summarizes the memory functions of the string-handling library. In the function discussions, “object” refers to a block of data.

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void *memcpy( void *s1, const void *s2, size_t n );</code></td>
<td>Copies <code>n</code> characters from the object pointed to by <code>s2</code> into the object pointed to by <code>s1</code>. A pointer to the resulting object is returned.</td>
</tr>
<tr>
<td><code>void *memmove( void *s1, const void *s2, size_t n );</code></td>
<td>Copies <code>n</code> characters from the object pointed to by <code>s2</code> into the object pointed to by <code>s1</code>. The copy is performed as if the characters were first copied from the object pointed to by <code>s2</code> into a temporary array and then from the temporary array into the object pointed to by <code>s1</code>. A pointer to the resulting object is returned.</td>
</tr>
<tr>
<td><code>int memcmp( const void *s1, const void *s2, size_t n );</code></td>
<td>Compares the first <code>n</code> characters of the objects pointed to by <code>s1</code> and <code>s2</code>. The function returns 0, less than 0 or greater than 0 if <code>s1</code> is equal to, less than or greater than <code>s2</code>.</td>
</tr>
<tr>
<td><code>void *memchr( const void *s, int c, size_t n );</code></td>
<td>Locates the first occurrence of <code>c</code> (converted to unsigned char) in the first <code>n</code> characters of the object pointed to by <code>s</code>. If <code>c</code> is found, a pointer to <code>c</code> in the object is returned. Otherwise, <code>NULL</code> is returned.</td>
</tr>
<tr>
<td><code>void *memset( void *s, int c, size_t n );</code></td>
<td>Copies <code>c</code> (converted to unsigned char) into the first <code>n</code> characters of the object pointed to by <code>s</code>. A pointer to the result is returned.</td>
</tr>
</tbody>
</table>

**Fig. 8.30** | Memory functions of the string-handling library.
The pointer parameters to these functions are declared \texttt{void *} so they can be used to manipulate memory for any data type. In Chapter 7, we saw that a pointer to any data type can be assigned directly to a pointer of type \texttt{void *}, and a pointer of type \texttt{void *} can be assigned directly to a pointer to any data type. For this reason, these functions can receive pointers to any data type. Because a \texttt{void *} pointer cannot be dereferenced, each function receives a size argument that specifies the number of characters (bytes) the function will process. For simplicity, the examples in this section manipulate character arrays (blocks of characters).

\textit{Function \texttt{memcpy}}

Function \texttt{memcpy} copies a specified number of characters from the object pointed to by its second argument into the object pointed to by its first argument. The function can receive a pointer to any type of object. The result of this function is undefined if the two objects overlap in memory (i.e., if they are parts of the same object)—in such cases, use \texttt{memmove}. Figure 8.31 uses \texttt{memcpy} to copy the string in array \texttt{s2} to array \texttt{s1}.

```c
/* Fig. 8.31: fig08_31.c */
#include <stdio.h>
#include <string.h>

int main( void )
{
    char s1[17]; /* create char array s1 */
    char s2[] = "Copy this string"; /* initialize char array s2 */

    memcpy( s1, s2, 17 );

    printf( "%s\n%s\n%s\n", "After s2 is copied into s1 with memcpy,",
    "s1 contains ", s1 );

    return 0; /* indicates successful termination */
}
```

\texttt{After s2 is copied into s1 with memcpy, s1 contains "Copy this string"}

\textit{Fig. 8.31} | Using \texttt{memcpy}.

\textit{Function \texttt{memmove}}

Function \texttt{memmove}, like \texttt{memcpy}, copies a specified number of bytes from the object pointed to by its second argument into the object pointed to by its first argument. Copying is performed as if the bytes were copied from the second argument into a temporary character array, then copied from the temporary array into the first argument. This allows characters from one part of a string to be copied into another part of the same string. Figure 8.32 uses \texttt{memmove} to copy the last 10 bytes of array \texttt{x} into the first 10 bytes of array \texttt{x}.

\textit{Common Programming Error 8.7}

String-manipulation functions other than \texttt{memmove} that copy characters have undefined results when copying takes place between parts of the same string.
8.9 Memory Functions of the String-Handling Library

**Function memcmp**

Function `memcmp` (Fig. 8.33) compares the specified number of characters of its first argument with the corresponding characters of its second argument. The function returns a value greater than 0 if the first argument is greater than the second, returns 0 if the arguments are equal and returns a value less than 0 if the first argument is less than the second.

```c
/* Fig. 8.33: fig08_33.c */
#include <stdio.h>
#include <string.h>

int main( void )
{
    char s1[] = "ABCDEFG"; /* initialize char array s1 */
    char s2[] = "ABCDXYZ"; /* initialize char array s2 */

    printf( "s1 = %s\n", s1 );
    printf( "s2 = %s\n", s2 );
    "memcmp( s1, s2, 4 ) = ", memcmp( s1, s2, 4 ),
    "memcmp( s1, s2, 7 ) = ", memcmp( s1, s2, 7 ),
    "memcmp( s2, s1, 7 ) = ", memcmp( s2, s1, 7 );
    return 0; /* indicate successful termination */
} /* end main */
```

Fig. 8.33 | Using memcmp.

s1 = ABCDEFG
s2 = ABCDXYZ

memcmp( s1, s2, 4 ) = 0
memcmp( s1, s2, 7 ) = -1
memcmp( s2, s1, 7 ) = 1
**Function memchr**

Function `memchr` searches for the first occurrence of a byte, represented as unsigned char, in the specified number of bytes of an object. If the byte is found, a pointer to the byte in the object is returned; otherwise, a NULL pointer is returned. Figure 8.34 searches for the character (byte) 'r' in the string "This is a string".

```c
/* Fig. 8.34: fig08_34.c */
#include <stdio.h>
#include <string.h>

int main( void )
{
    const char *s = "This is a string"; /* initialize char pointer */
    printf( "%s\%c\%s"\"\n",
            "The remainder of s after character ", 'r',
            " is found is ", memchr( s, 'r', 16 ) );
    return 0; /* indicates successful termination */
} /* end main */
```

The remainder of s after character 'r' is found is "ring"

---

**Function memset**

Function `memset` copies the value of the byte in its second argument into the first \( n \) bytes of the object pointed to by its first argument, where \( n \) is specified by the third argument. Figure 8.35 uses memset to copy 'b' into the first 7 bytes of `string1`.

```c
/* Fig. 8.35: fig08_35.c */
#include <stdio.h>
#include <string.h>

int main( void )
{
    char string1[ 15 ] = "BBBBBBBBBBBBB"; /* initialize string1 */
    printf( "string1 = %s\n", string1 );
    printf( "string1 after memset = %s\n", memset( string1, 'b', 7 ) );
    return 0; /* indicates successful termination */
} /* end main */
```

string1 = BBBBBBBBBBBBBB
string1 after memset = bbbbbbbbbBBBBB

---

Fig. 8.34 | Using `memchr`.

Fig. 8.35 | Using `memset`. 
8.10 Other Functions of the String-Handling Library

The two remaining functions of the string-handling library are `strerror` and `strlen`. Figure 8.36 summarizes the `strerror` and `strlen` functions.

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char *strerror( int errornum );</code></td>
<td>Maps <code>errornum</code> into a full text string in a compiler- and locale-specific manner (e.g., the message may appear in different languages based on its location). A pointer to the string is returned.</td>
</tr>
<tr>
<td><code>size_t strlen( const char *s );</code></td>
<td>Determines the length of string <code>s</code>. The number of characters preceding the terminating null character is returned.</td>
</tr>
</tbody>
</table>

**Fig. 8.36** | Other functions of the string-handling library.

*Function `strerror`*

Function `strerror` takes an error number and creates an error message string. A pointer to the string is returned. Figure 8.37 demonstrates `strerror`.

```c
/* Fig. 8.37: fig08_37.c */
#include <stdio.h>
#include <string.h>

int main( void )
{
    printf( "%s\n", strerror( 2 ) );
    return 0; /* indicates successful termination */
} /* end main */
```

No such file or directory

**Fig. 8.37** | Using `strerror`.

*Function `strlen`*

Function `strlen` takes a string as an argument and returns the number of characters in the string—the terminating null character is not included in the length. Figure 8.38 demonstrates function `strlen`.

```c
/* Fig. 8.38: fig08_38.c */
#include <stdio.h>
#include <string.h>

/* Fig. 8.38 | Using `strlen`. (Part 1 of 2.) */
```
int main( void )
{
    /* initialize 3 char pointers */
    const char *string1 = "abcdefghijklmnopqrstuvwxyz";
    const char *string2 = "four";
    const char *string3 = "Boston";

    printf("The length of "%s" is %lu\n", string1, (unsigned long) strlen( string1 ) );
    printf("The length of "%s" is %lu\n", string2, (unsigned long) strlen( string2 ) );
    printf("The length of "%s" is %lu\n", string3, (unsigned long) strlen( string3 ) );

    return 0; /* indicates successful termination */
} /* end main */
• Function `isdigit` determines whether its argument is a digit (0–9).
• Function `isalpha` determines whether its argument is an uppercase letter (A–Z) or a lowercase letter (a–z).
• Function `isalnum` determines whether its argument is an uppercase letter (A–Z), a lowercase letter (a–z) or a digit (0–9).
• Function `isxdigit` determines whether its argument is a hexadecimal digit (A–F, a–f, 0–9).
• Function `toupper` converts a lowercase letter to uppercase and returns the uppercase letter.
• Function `tolower` converts an uppercase letter to lowercase and returns the lowercase letter.
• Function `isspace` determines whether its argument is one of the following white-space characters: ' ' (space), '', '
', '', '	' or '\v'.
• Function `iscntrl` determines whether its argument is one of the following control characters: '	', '\v', '', '\a', '\b', '\r' or '\n'.
• Function `ispunct` determines whether its argument is a printing character other than a space, a digit or a letter.
• Function `isprint` determines whether its argument is any printing character including the space character.
• Function `isgraph` determines whether its argument is a printing character other than the space character.

**Section 8.4 String-Conversion Functions**
• Function `atof` converts its argument—a string beginning with a series of digits that represents a floating-point number—to a `double` value.
• Function `atoi` converts its argument—a string beginning with a series of digits that represents an integer—to an `int` value.
• Function `atol` converts its argument—a string beginning with a series of digits that represents a long integer—to a `long` value.
• Function `strtod` converts a sequence of characters representing a floating-point value to `double`. The function receives two arguments—a string (char *) and a pointer to char *. The string contains the character sequence to be converted, and the location specified by the pointer to char * is assigned the address of the remainder of the string after the conversion.
• Function `strtol` converts a sequence of characters representing an integer to `long`. The function receives three arguments—a string (char *), a pointer to char * and an integer. The string contains the character sequence to be converted, the location specified by the pointer to char * is assigned the address of the remainder of the string after the conversion and the integer specifies the base of the value being converted.
• Function `strtoul` converts a sequence of characters representing an integer to `unsigned long`. The function receives three arguments—a string (char *), a pointer to char * and an integer. The string contains the character sequence to be converted, the location specified by the pointer to char * is assigned the address of the remainder of the string after the conversion and the integer specifies the base of the value being converted.

**Section 8.5 Standard Input/Output Library Functions**
• Function `fgets` reads characters until a newline character or the end-of-file indicator is encountered. The arguments to `fgets` are an array of type char, the maximum number of characters that can be read and the stream from which to read. A null character (‘\0’) is appended to the array after reading terminates.
• Function `putchar` prints its character argument.
• Function `getchar` reads a single character from the standard input and returns the character as an integer. If the end-of-file indicator is encountered, `getchar` returns `EOF`.

• Function `puts` takes a string (`char *`) as an argument and prints the string followed by a newline character.

• Function `sprintf` uses the same conversion specifications as function `printf` to print formatted data into an array of type `char`.

• Function `sscanf` uses the same conversion specifications as function `scanf` to read formatted data from a string.

Section 8.6 String-Manipulation Functions of the String-Handling Library

• Function `strcpy` copies its second argument (a string) into its first argument (a character array). You must ensure that the array is large enough to store the string and its terminating null character.

• Function `strncpy` is equivalent to `strcpy`, except that a call to `strncpy` specifies the number of characters to be copied from the string into the array. The terminating null character will be copied only if the number of characters to be copied is one more than the length of the string.

• Function `strcat` appends its second string argument—including the terminating null character—to its first string argument. The first character of the second string replaces the null (`'\0'`) character of the first string. You must ensure that the array used to store the first string is large enough to store both the first string and the second string.

• Function `strncat` appends a specified number of characters from the second string to the first string. A terminating null character is appended to the result.

Section 8.7 Comparison Functions of the String-Handling Library

• Function `strcmp` compares its first string argument to its second string argument, character by character. It returns 0 if the strings are equal, returns a negative value if the first string is less than the second and returns a positive value if the first string is greater than the second.

• Function `strncmp` is equivalent to `strcmp`, except that `strncmp` compares a specified number of characters. If one of the strings is shorter than the number of characters specified, `strncmp` compares characters until the null character in the shorter string is encountered.

Section 8.8 Search Functions of the String-Handling Library

• Function `strchr` searches for the first occurrence of a character in a string. If the character is found, `strchr` returns a pointer to the character in the string; otherwise, `strchr` returns `NULL`.

• Function `strcspn` determines the length of the initial part of the string in its first argument that does not contain any characters from the string in its second argument. The function returns the length of the segment.

• Function `strpbrk` searches for the first occurrence in its first argument of any character in its second argument. If a character from the second argument is found, `strpbrk` returns a pointer to the character; otherwise, `strpbrk` returns `NULL`.

• Function `strrchr` searches for the last occurrence of a character in a string. If the character is found, `strrchr` returns a pointer to the character in the string; otherwise, `strrchr` returns `NULL`.

• Function `strspn` determines the length of the initial part of the string in its first argument that contains only characters from the string in its second argument. The function returns the length of the segment.

• Function `strstr` searches for the first occurrence of its second string argument in its first string argument. If the second string is found in the first string, a pointer to the location of the string in the first argument is returned.
A sequence of calls to `strtok` breaks the first string `s1` into tokens that are separated by characters contained in the second string `s2`. The first call contains `s1` as the first argument, and subsequent calls to continue tokenizing the same string contain `NULL` as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, a `NULL` pointer is returned.

### Section 8.9 Memory Functions of the String-Handling Library

- Function `memcpy` copies a specified number of characters from the object to which its second argument points into the object to which its first argument points. The function can receive a pointer to any type of object. Function `memcpy` manipulates the bytes of the object as characters.

- Function `memmove` copies a specified number of bytes from the object pointed to by its second argument to the object pointed to by its first argument. Copying is accomplished as if the bytes were copied from the second argument to a temporary character array and then copied from the temporary array to the first argument.

- Function `memcmp` compares the specified number of characters of its first and second arguments.

- Function `memchr` searches for the first occurrence of a byte, represented as `unsigned char`, in the specified number of bytes of an object. If the byte is found, a pointer to the byte is returned; otherwise, a `NULL` pointer is returned.

- Function `memset` copies its second argument, treated as an `unsigned char`, to a specified number of bytes of the object pointed to by the first argument.

### Section 8.10 Other Functions of the String-Handling Library

- Function `strerror` maps an integer error number into a full text string in a locale specific manner. A pointer to the string is returned.

- Function `strlen` takes a string as an argument and returns the number of characters in the string—the terminating null character is not included in the length of the string.

### Terminology

- ASCII (American Standard Code for Information Interchange) 330
- `atof` function 318
- `atoi` function 318
- `atol` function 319
- Character constant 310
- Character-handling library 312
- Character set 310
- Comparing string 326
- Concatenating strings 326
- Control characters 316
- Copying string 326
- Delimiter 336
- Determining the length of string 326
- EBCDIC (Extended Binary Coded Decimal Interchange Code) 330
- `fgets` function 323
- General utilities library (`<stdlib.h>`) 317
- `getchar` function 324
- Hexadecimal 313
- `isalnum` function 313
- `isalpha` function 313
- `iscntrl` function 316
- `isdigit` function 313
- `isgraph` function 316
- `islower` function 314
- `isprint` function 316
- `ispunct` function 316
- `isspace` function 316
- `isupper` function 314
- `isxdigit` function 313
- `memchr` function 340
- `memcmp` function 339
- `memcpy` function 338
- `memmove` function 338
- `memset` function 340
- `null character` (`'\0'`) 310
- Numeric code 330
- Printing character 316
- `putchar` function 323
- `puts` function 324
- Special character 310

Terminology
Self-Review Exercises

8.1 Write a single statement to accomplish each of the following. Assume that variables c (which stores a character), x, y and z are of type int, variables d, e and f are of type double, variable ptr is of type char * and arrays s1[100] and s2[100] are of type char.

a) Convert the character stored in variable c to an uppercase letter. Assign the result to variable c.

b) Determine if the value of variable c is a digit. Use the conditional operator as shown in Figs. 8.2–8.4 to print "is a " or " is not a " when the result is displayed.

c) Convert the string "1234567" to long and print the value.

d) Determine if the value of variable c is a control character. Use the conditional operator to print " is a " or " is not a " when the result is displayed.

e) Read a line of text into array s1 from the keyboard. Do not use scanf.

f) Print the line of text stored in array s1. Do not use printf.

g) Assign ptr the location of the last occurrence of c in s1.

h) Print the value of variable c. Do not use printf.

i) Convert the string "8.63582" to double and print the value.

j) Determine if the value of c is a letter. Use the conditional operator to print " is a " or " is not a " when the result is displayed.

k) Read a character from the keyboard and store the character in variable c.

l) Assign ptr the location of the first occurrence of s2 in s1.

m) Determine if the value of variable c is a printing character. Use the conditional operator to print " is a " or " is not a " when the result is displayed.

n) Read three double values into variables d, e and f from the string "1.27 10.3 9.432".

o) Copy the string stored in array s2 into array s1.

p) Assign ptr the location of the first occurrence in s1 of any character from s2.

q) Compare the string in s1 with the string in s2. Print the result.

r) Assign ptr the location of the first occurrence of c in s1.

s) Use sprintf to print the values of integer variables x, y and z into array s1. Each value should be printed with a field width of 7.

t) Append 10 characters from the string in s2 to the string in s1.

u) Determine the length of the string in s1. Print the result.

v) Convert the string "-21" to int and print the value.

w) Assign ptr to the location of the first token in s2. Tokens in the string s2 are separated by commas (,).
8.2  Show two different methods of initializing character array \texttt{vowel} with the string of vowels "AEIOU".

8.3  What, if anything, prints when each of the following C statements is performed? If the statement contains an error, describe the error and indicate how to correct it. Assume the following variable definitions:

\begin{verbatim}
char s1[50] = "jack", s2[50] = "jill", s3[50], *sptr;
\end{verbatim}

\begin{enumerate}
\item \texttt{printf( \textasciitilde%c\textasciitilde%s\textasciitilde, toupper( s1[0]), &s1[1]);}
\item \texttt{printf( \textasciitilde%s\textasciitilde, strcpy( s3, s2 ));}
\item \texttt{printf( \textasciitilde%s\textasciitilde, strcat( strcat( strcpy( s3, s1 ), " and " ), s2 ));}
\item \texttt{printf( \textasciitilde%u\textasciitilde, strlen(s1) + strlen(s2));}
\item \texttt{printf( \textasciitilde%u\textasciitilde, strlen(s3));}
\end{enumerate}

8.4  Find the error in each of the following program segments and explain how to correct it:

\begin{enumerate}
\item \texttt{char s[10];
strncpy( s, \textquote{hello}, 5 );
printf( \textquote{\%sz}, s );}
\item \texttt{printf( \textquote{\%sz}, \textquote{a});}
\item \texttt{char s[12];
strcpy( s, \textquote{Welcome Home});}
\item \texttt{if ( strcmp( string1, string2 ) ) {
    printf( \textquote{The strings are equal\n});
}}
\end{enumerate}

\textbf{Answers to Self-Review Exercises}

8.1  \begin{enumerate}
\item \texttt{c = toupper( c );}
\item \texttt{printf( \textquote{\%c\textasciitilde%digit\n}, c, isdigit( c ) ? \textquote{is a } : \textquote{is not a });}
\item \texttt{printf( \textquote{\%ld\n}, atol( \textquote{1234567});}
\item \texttt{printf( \textquote{\%c\textasciitilde%control character\n}, c, iscntrl( c ) ? \textquote{is a } : \textquote{is not a });}
\item \texttt{fgets( s1, 100, stdin );}
\item \texttt{puts( s1 );}
\item \texttt{ptr = strrchr( s1, c );}
\item \texttt{putchar( c );}
\item \texttt{printf( \textquote{\%f\n}, atof( \textquote{8.63582});}
\item \texttt{printf( \textquote{\%c\textasciitilde%letter\n}, c, isalpha( c ) ? \textquote{is a } : \textquote{is not a });}
\item \texttt{c = getchar();}
\item \texttt{ptr = strstr( s1, s2 );}
\item \texttt{printf( \textquote{\%c\textasciitilde%sprinting character\n}, c, isprint( c ) ? \textquote{is a } : \textquote{is not a });}
\item \texttt{sscanf( \textquote{1.27 10.3 9.432}, \textquote{\%f\%f\%f}, \&d, \&e, \&f );}
\item \texttt{strcpy( s1, s2 );}
\item \texttt{ptr = strpbrk( s1, s2 );}
\item \texttt{printf( \textquote{strcmp( s1, s2 ) = \%d\n}, strcmp( s1, s2 ));}
\item \texttt{ptr = strchr( s1, c );}
\item \texttt{printf( \textquote{\%d\n}, atoi( \textquote{-21});}
\item \texttt{ptr = strtok( s2, \textquote{,});}
\end{enumerate}
8.2  char vowel[] = "AEIOU";
     char vowel[] = {'A', 'E', 'I', 'O', 'U', '\0'};

8.3  a) Jack
     b) jill
     c) jack and jill
     d) 8
     e) 13

8.4  a) Error: Function strncpy does not write a terminating null character to array s, because its third argument is equal to the length of the string "hello". Correction: Make the third argument of strncpy 6, or assign '\0' to s[5].
     b) Error: Attempting to print a character constant as a string. Correction: Use %c to output the character, or replace 'a' with "a".
     c) Error: Character array s is not large enough to store the terminating null character. Correction: Declare the array with more elements.
     d) Error: Function strcmp returns 0 if the strings are equal; therefore, the condition in the if statement is false, and the printf will not be executed. Correction: Compare the result of strcmp with 0 in the condition.

Exercises

8.5  (Character Testing) Write a program that inputs a character from the keyboard and tests the character with each of the functions in the character-handling library. The program should print the value returned by each function.

8.6  (Displaying Strings in Uppercase and Lowercase) Write a program that inputs a line of text into char array s[100]. Output the line in uppercase letters and in lowercase letters.

8.7  (Converting Strings to Integers for Calculations) Write a program that inputs four strings that represent integers, converts the strings to integers, sums the values and prints the total of the four values.

8.8  (Converting Strings to Floating Point for Calculations) Write a program that inputs four strings that represent floating-point values, converts the strings to double values, sums the values and prints the total of the four values.

8.9  (Comparing Strings) Write a program that uses function strcmp to compare two strings input by the user. The program should state whether the first string is less than, equal to or greater than the second string.

8.10 (Comparing Portions of Strings) Write a program that uses function strncmp to compare two strings input by the user. The program should input the number of characters to be compared, then display whether the first string is less than, equal to or greater than the second string.

8.11 (Random Sentences) Write a program that uses random number generation to create sentences. The program should use four arrays of pointers to char called article, noun, verb and preposition. The program should create a sentence by selecting a word at random from each array in the following order: article, noun, verb, preposition, article and noun. As each word is picked, it should be concatenated to the previous words in an array large enough to hold the entire sentence. The words should be separated by spaces. When the final sentence is output, it should start with a capital letter and end with a period. The program should generate 20 such sentences. The arrays should be filled as follows: The article array should contain the articles "the", "a", "one", "some" and "any"; the noun array should contain the nouns "boy", "girl", "dog", "town" and "car"; the verb array should contain the verbs "drove", "jumped", "ran", "walked" and "skipped"; the preposition array should contain the prepositions "to", "from", "over", "under" and "on".
After the preceding program is written and working, modify it to produce a short story consisting of several of these sentences. (How about the possibility of a random term paper writer?)

8.12 **(Limericks)** A limerick is a humorous five-line verse in which the first and second lines rhyme with the fifth, and the third line rhymes with the fourth. Using techniques similar to those developed in Exercise 8.11, write a program that produces random limericks. Polishing this program to produce good limericks is a challenging problem, but the result will be worth the effort!

8.13 **(Pig Latin)** Write a program that encodes English-language phrases into pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig-Latin phrases. For simplicity, use the following algorithm:

To form a pig-Latin phrase from an English-language phrase, tokenize the phrase into words with function `strtok`. To translate each English word into a pig-Latin word, place the first letter of the English word at the end of the English word and add the letters “ay.” Thus the word “jump” becomes “umpjay,” the word “the” becomes “hetay” and the word “computer” becomes “omputer-cay.” Blanks between words remain as blanks. Assume the following: The English phrase consists of words separated by blanks, there are no punctuation marks, and all words have two or more letters. Function `printLatinWord` should display each word. [ Hint: Each time a token is found in a call to `strtok`, pass the token pointer to function `printLatinWord`, and print the pig-Latin word.]

8.14 **(Tokenizing Telephone Numbers)** Write a program that inputs a telephone number as a string in the form (555) 555-5555. The program should use function `strtok` to extract the area code as a token, the first three digits of the phone number as a token and the last four digits of the phone number as a token. The seven digits of the phone number should be concatenated into one string. The program should convert the area-code string to `int` and convert the phone-number string to `long`. Both the area code and the phone number should be printed.

8.15 **(Displaying a Sentence with Its Words Reversed)** Write a program that inputs a line of text, tokenizes the line with function `strtok` and outputs the tokens in reverse order.

8.16 **(Searching for Substrings)** Write a program that inputs a line of text and a search string from the keyboard. Using function `strstr`, locate the first occurrence of the search string in the line of text, and assign the location to variable `searchPtr` of type `char*`. If the search string is found, print the remainder of the line of text beginning with the search string. Then, use `strstr` again to locate the next occurrence of the search string in the line of text. If a second occurrence is found, print the remainder of the line of text beginning with the second occurrence. [Hint: The second call to `strstr` should contain `searchPtr + 1` as its first argument.]

8.17 **(Counting the Occurrences of a Substring)** Write a program based on the program of Exercise 8.16 that inputs several lines of text and a search string and uses function `strstr` to determine the total occurrences of the string in the lines of text. Print the result.

8.18 **(Counting the Occurrences of a Character)** Write a program that inputs several lines of text and a search character and uses function `strchr` to determine the total occurrences of the character in the lines of text.

8.19 **(Counting the Letters of the Alphabet in a String)** Write a program based on the program of Exercise 8.18 that inputs several lines of text and uses function `strchr` to determine the total occurrences of each letter of the alphabet in the lines of text. Uppercase and lowercase letters should be counted together. Store the totals for each letter in an array and print the values in tabular format after the totals have been determined.

8.20 **(Counting the Number of Words in a String)** Write a program that inputs several lines of text and uses `strtok` to count the total number of words. Assume that the words are separated by either spaces or newline characters.
8.21 (Alphabetizing a List of Strings) Use the string-comparison functions discussed in Section 8.6 and the techniques for sorting arrays developed in Chapter 6 to write a program that alphabetizes a list of strings. Use the names of 10 or 15 towns in your area as data for your program.

8.22 The chart in Appendix B shows the numeric code representations for the characters in the ASCII character set. Study this chart and then state whether each of the following is true or false.
   a) The letter “A” comes before the letter “B.”
   b) The digit “9” comes before the digit “0.”
   c) The commonly used symbols for addition, subtraction, multiplication and division all come before any of the digits.
   d) The digits come before the letters.
   e) If a sort program sorts strings into ascending sequence, then the program will place the symbol for a right parenthesis before the symbol for a left parenthesis.

8.23 (Strings Starting with “b”) Write a program that reads a series of strings and prints only those strings beginning with the letter “b.”

8.24 (Strings Ending with “ed”) Write a program that reads a series of strings and prints only those strings that end with the letters “ed.”

8.25 (Printing Letters for Various ASCII Codes) Write a program that inputs an ASCII code and prints the corresponding character. Modify this program so that it generates all possible three-digit codes in the range 000 to 255 and attempts to print the corresponding characters. What happens when this program is run?

8.26 (Write Your Own Character-Handling Functions) Using the ASCII character chart in Appendix B as a guide, write your own versions of the character-handling functions in Fig. 8.1.

8.27 (Write Your String Conversion Functions) Write your own versions of the functions in Fig. 8.5 for converting strings to numbers.

8.28 (Write Your Own String Copy and Concatenation Functions) Write two versions of each of the string-copy and string-concatenation functions in Fig. 8.17. The first version should use array subscripting, and the second version should use pointers and pointer arithmetic.

8.29 (Write Your Own Character and String I/O Functions) Write your own versions of the functions getchar, putchar and puts described in Fig. 8.12.

8.30 (Write Your Own String Comparison Functions) Write two versions of each string-comparison function in Fig. 8.20. The first version should use array subscripting, and the second version should use pointers and pointer arithmetic.

8.31 (Write Your Own String Searching Functions) Write your own versions of the functions in Fig. 8.22 for searching strings.

8.32 (Write Your Own Memory-Handling Functions) Write your own versions of the functions in Fig. 8.30 for manipulating blocks of memory.

8.33 (Write Your Own String Length Function) Write two versions of function strlen in Fig. 8.36. The first version should use array subscripting, and the second version should use pointers and pointer arithmetic.

Special Section: Advanced String-Manipulation Exercises

The preceding exercises are keyed to the text and designed to test the reader’s understanding of fundamental string-manipulation concepts. This section includes a collection of intermediate and advanced problems. The reader should find these problems challenging yet enjoyable. The problems vary considerably in difficulty. Some require an hour or two of program writing and
implementation. Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects.

8.34 (Text Analysis) The availability of computers with string-manipulation capabilities has resulted in some rather interesting approaches to analyzing the writings of great authors. Much attention has been focused on whether William Shakespeare ever lived. Some scholars find substantial evidence that Christopher Marlowe actually penned the masterpieces attributed to Shakespeare. Researchers have used computers to find similarities in the writings of these two authors. This exercise examines three methods for analyzing texts with a computer.

a) Write a program that reads several lines of text and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the phrase

To be, or not to be: that is the question:

contains one “a,” two “b’s,” no “c’s,” and so on.

b) Write a program that reads several lines of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, and so on, appearing in the text. For example, the phrase

Whether 'tis nobler in the mind to suffer

contains

<table>
<thead>
<tr>
<th>Word length</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2 (including 'tis)</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

c) Write a program that reads several lines of text and prints a table indicating the number of occurrences of each different word in the text. The first version of your program should include the words in the table in the same order in which they appear in the text. A more interesting (and useful) printout should then be attempted in which the words are sorted alphabetically. For example, the lines

To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer

contain the words “to” three times, the word “be” two times, the word “or” once, and so on.

8.35 (Word Processing) The detailed treatment of string manipulation in this text is largely attributable to the exciting growth in word processing in recent years. One important function in word-processing systems is type justification—the alignment of words to both the left and right margins of a page. This generates a professional-looking document that gives the appearance of being set in type, rather than prepared on a typewriter. Type justification can be accomplished on computer systems by inserting one or more blank characters between each of the words in a line so that the rightmost word aligns with the right margin.

Write a program that reads several lines of text and prints this text in type-justified format. Assume that the text is to be printed on 8 1/2-inch-wide paper and that one-inch margins are to be
allowed on both the left and right sides of the printed page. Assume that the computer prints 10 characters to the horizontal inch. Therefore, your program should print 6 1/2 inches of text or 65 characters per line.

8.36 (Printing Dates in Various Formats) Dates are commonly printed in several different formats in business correspondence. Two of the more common formats are

07/21/2003 and July 21, 2003

Write a program that reads a date in the first format and prints it in the second format.

8.37 (Check Protection) Computers are frequently used in check-writing systems, such as payroll and accounts payable applications. Many stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of $1 million. Weird amounts are printed by computerized check-writing systems because of human error and/or machine failure. Systems designers, of course, make every effort to build controls into their systems to prevent erroneous checks from being issued.

Another serious problem is the intentional alteration of a check amount by someone who intends to cash it fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called check protection.

Checks designed for imprinting by computer contain a fixed number of spaces in which the computer may print an amount. Suppose a paycheck contains nine blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all nine of those spaces will be filled—for example:

```
11,230.60 (check amount)
---------
123456789 (position numbers)
```

On the other hand, if the amount is less than $1000, then several of the spaces will ordinarily be left blank—for example,

```
99.87
---------
123456789
```

contains four blank spaces. If a check is printed with blank spaces, it is easier for someone to alter the amount of the check. To prevent a check from being altered, many check-writing systems insert leading asterisks to protect the amount as follows:

```
****99.87
---------
123456789
```

Write a program that inputs a dollar amount to be printed on a check and then prints the amount in check-protected format with leading asterisks if necessary. Assume that nine spaces are available for printing an amount.

8.38 (Writing the Word Equivalent of a Check Amount) Continuing the discussion of the previous example, we reiterate the importance of designing check-writing systems to prevent alteration of check amounts. One common security method requires that the check amount be both written in numbers and “spelled out” in words. Even if someone is able to alter the numerical amount of the check, it is extremely difficult to change the amount in words.

Many computerized check-writing systems do not print the amount of the check in words. Perhaps the main reason for this omission is the fact that most high-level languages used in commercial applications do not contain adequate string-manipulation features. Another reason is that the logic for writing word equivalents of check amounts is somewhat involved.

Write a program that inputs a numeric check amount and writes the word equivalent of the amount. For example, the amount 112.43 should be written as

ONE HUNDRED TWELVE and 43/100
8.39 (Morse Code) Perhaps the most famous of all coding schemes is Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit, and a few special characters (such as period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound and the dash represents a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems.

Separation between words is indicated by a space—quite simply, the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short period of time during which no sound is transmitted. The international version of Morse code appears in Fig. 8.39.

<table>
<thead>
<tr>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.-</td>
<td>N</td>
<td>-.</td>
<td>Digits</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>-...</td>
<td>O</td>
<td>---</td>
<td>1</td>
<td>----</td>
</tr>
<tr>
<td>C</td>
<td>-.-</td>
<td>P</td>
<td>..-</td>
<td>2</td>
<td>----</td>
</tr>
<tr>
<td>D</td>
<td>-...</td>
<td>Q</td>
<td>--.</td>
<td>3</td>
<td>-----</td>
</tr>
<tr>
<td>E</td>
<td>.</td>
<td>R</td>
<td>.-.</td>
<td>4</td>
<td>......</td>
</tr>
<tr>
<td>F</td>
<td>--.</td>
<td>S</td>
<td>...</td>
<td>5</td>
<td>......</td>
</tr>
<tr>
<td>G</td>
<td>---</td>
<td>T</td>
<td>-</td>
<td>6</td>
<td>-....</td>
</tr>
<tr>
<td>H</td>
<td>.....</td>
<td>U</td>
<td>..-</td>
<td>7</td>
<td>----.</td>
</tr>
<tr>
<td>I</td>
<td>...</td>
<td>V</td>
<td>...-</td>
<td>8</td>
<td>----.</td>
</tr>
<tr>
<td>J</td>
<td>----</td>
<td>W</td>
<td>--.</td>
<td>9</td>
<td>-----</td>
</tr>
<tr>
<td>K</td>
<td>-.</td>
<td>X</td>
<td>-.--</td>
<td>0</td>
<td>-----</td>
</tr>
<tr>
<td>L</td>
<td>--.</td>
<td>Y</td>
<td>-.--</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>--</td>
<td>Z</td>
<td>--..</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 8.39 | The letters of the alphabet as expressed in international Morse code.

Write a program that reads an English-language phrase and encodes the phrase into Morse code. Also write a program that reads a phrase in Morse code and converts the phrase into the English-language equivalent. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word.

8.40 (A Metric Conversion Program) Write a program that will assist the user with metric conversions. Your program should allow the user to specify the names of the units as strings (i.e., centimeters, liters, grams, and so on for the metric system and inches, quarts, pounds, and so on for the English system) and should respond to simple questions such as

"How many inches are in 2 meters?"
"How many liters are in 10 quarts?"

Your program should recognize invalid conversions. For example, the question

"How many feet in 5 kilograms?"

is not meaningful, because "feet" are units of length while "kilograms" are units of mass.

8.41 (Dunning Letters) Many businesses spend a great deal of time and money collecting overdue debts. Dunning is the process of making repeated and insistent demands upon a debtor in an attempt to collect a debt.
Computers are often used to generate dunning letters automatically and in increasing degrees of severity as a debt ages. The theory is that as a debt becomes older, it becomes more difficult to collect, and therefore the dunning letters must become more threatening.

Write a program that contains the texts of five dunning letters of increasing severity. Your program should accept as input the following:

a) Debtor’s name
b) Debtor’s address
c) Debtor’s account
d) Amount owed
e) Age of the amount owed (i.e., one month overdue, two months overdue, and so on).

Use the age of the amount owed to select one of the five message texts, and then print the dunning letter, inserting the other user-supplied information where appropriate.

A Challenging String-Manipulation Project

8.42  (A Crossword-Puzzle Generator) Most people have worked a crossword puzzle at one time or another, but few have ever attempted to generate one. Generating a crossword puzzle is a difficult problem. It is suggested here as a string-manipulation project requiring substantial sophistication and effort. There are many issues you must resolve to get even the simplest crossword-puzzle generator program working. For example, how does one represent the grid of a crossword puzzle inside the computer? Should one use a series of strings, or should double-subscripted arrays be used? You need a source of words (i.e., a computerized dictionary) that can be directly referenced by the program. In what form should these words be stored to facilitate the complex manipulations required by the program? The really ambitious reader will want to generate the “clues” portion of the puzzle in which the brief hints for each “across” word and each “down” word are printed for the puzzle worker. Merely printing a version of the blank puzzle itself is not a simple problem.

Making a Difference

8.43  (Cooking with Healthier Ingredients) Obesity in America is increasing at an alarming rate. Check the map from the Centers for Disease Control and Prevention (CDC) at www.cdc.gov/nccdphp/dnpa/obesity/trend/maps/index.htm, which shows obesity trends in the United States over the last 20 years. As obesity increases, so do occurrences of related problems (e.g., heart disease, high blood pressure, high cholesterol, type 2 diabetes). Write a program that helps users choose healthier ingredients when cooking, and helps those allergic to certain foods (e.g., nuts, gluten) find substitutes. The program should read a recipe from the user and suggest healthier replacements for some of the ingredients. For simplicity, your program should assume the recipe has no abbreviations for measures such as teaspoons, cups, and tablespoons, and uses numerical digits for quantities (e.g., 1 egg, 2 cups) rather than spelling them out (one egg, two cups). Some common substitutions are shown in Fig. 8.40. Your program should display a warning such as, “Always consult your physician before making significant changes to your diet.”

Your program should take into consideration that replacements are not always one-for-one. For example, if a cake recipe calls for three eggs, it might reasonably use six egg whites instead. Conversion data for measurements and substitutes can be obtained at websites such as:

chinesefood.about.com/od/recipeconversionfaqs/f/usmetricrecipes.htm
www.pioneerthinking.com/eggsub.html
www.gourmetsleuth.com/conversions.htm

Your program should consider the user’s health concerns, such as high cholesterol, high blood pressure, weight loss, gluten allergy, and so on. For high cholesterol, the program should suggest substitutes for eggs and dairy products; if the user wishes to lose weight, low-calorie substitutes for ingredients such as sugar should be suggested.
8.44  *(Spam Scanner)* Spam (or junk e-mail) costs U.S. organizations billions of dollars a year in spam-prevention software, equipment, network resources, bandwidth, and lost productivity. Research online some of the most common spam e-mail messages and words, and check your own junk e-mail folder. Create a list of 30 words and phrases commonly found in spam messages. Write a program in which the user enters an e-mail message. Read the message into a large character array and ensure that the program does not attempt to insert characters past the end of the array. Then, scan the message for each of the 30 keywords or phrases. For each occurrence of one of these within the message, add a point to the message’s “spam score.” Next, rate the likelihood that the message is spam, based on the number of points it received.

8.45  *(SMS Language)* Short Message Service (SMS) is a communications service that allows sending text messages of 160 or fewer characters between mobile phones. With the proliferation of mobile phone use worldwide, SMS is being used in many developing nations for political purposes (e.g., voicing opinions and opposition), reporting news about natural disasters, and so on. For example, check out comunica.org/radio2.0/archives/87. Since the length of SMS messages is limited, SMS Language—abbreviations of common words and phrases in mobile text messages, e-mails, instant messages, etc.—is often used. For example, “in my opinion” is “IMO” in SMS Language. Research SMS Language online. Write a program in which the user can enter a message using SMS Language, then the program translates it into English (or your own language). Also provide a mechanism to translate text written in English (or your own language) into SMS Language. One potential problem is that one SMS abbreviation could expand into a variety of phrases. For example, IMO (as used above) could also stand for “International Maritime Organization,” “in memory of,” etc.

8.46  *(Gender-Neutrality)* In Exercise 1.12, you researched eliminating sexism in all forms of communication. You then described the algorithm you’d use to read through a paragraph of text and replace gender-specific words with gender-neutral equivalents. Create a program that reads a paragraph of text, then replaces gender-specific words with gender-neutral ones. Display the resulting gender-neutral text.

---

**Fig. 8.40**  Typical ingredient substitutes.

<table>
<thead>
<tr>
<th>Ingredient</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 cup sour cream</td>
<td>1 cup yogurt</td>
</tr>
<tr>
<td>1 cup milk</td>
<td>1/2 cup evaporated milk and 1/2 cup water</td>
</tr>
<tr>
<td>1 teaspoon lemon juice</td>
<td>1/2 teaspoon vinegar</td>
</tr>
<tr>
<td>1 cup sugar</td>
<td>1/2 cup honey, 1 cup molasses or 1/4 cup agave nectar</td>
</tr>
<tr>
<td>1 cup butter</td>
<td>1 cup margarine or yogurt</td>
</tr>
<tr>
<td>1 cup flour</td>
<td>1 cup rye or rice flour</td>
</tr>
<tr>
<td>1 cup mayonnaise</td>
<td>1 cup cottage cheese or 1/8 cup mayonnaise and 7/8 cup yogurt</td>
</tr>
<tr>
<td>1 egg</td>
<td>2 tablespoons cornstarch, arrowroot flour or potato starch or 2 egg whites or 1/2 of a large banana (mashed)</td>
</tr>
<tr>
<td>1/2 cup oil</td>
<td>1/4 cup applesauce</td>
</tr>
<tr>
<td>white bread</td>
<td>whole-grain bread</td>
</tr>
</tbody>
</table>
Objectives
In this chapter, you’ll learn:

- To use input and output streams.
- To use all print formatting capabilities.
- To use all input formatting capabilities.
- To print with field widths and precisions.
- To use formatting flags in the `printf` format control string.
- To output literals and escape sequences.
- To format input using `scanf`.

All the news that’s fit to print.
—Adolph S. Ochs

What mad pursuit? What struggle to escape?
—John Keats

Remove not the landmark on the boundary of the fields.
—Amenemope
9.1 Introduction

An important part of the solution to any problem is the presentation of the results. In this chapter, we discuss in depth the formatting features of `scanf` and `printf`. These functions input data from the standard input stream and output data to the standard output stream. Four other functions that use the standard input and standard output—`gets`, `puts`, `getchar` and `putchar`—were discussed in Chapter 8. Include the header `<stdio.h>` in programs that call these functions.

Many features of `printf` and `scanf` were discussed earlier in the text. This chapter summarizes those features and introduces others. Chapter 11 discusses several additional functions included in the standard input/output (<stdio.h>) library.

9.2 Streams

All input and output is performed with streams, which are sequences of bytes. In input operations, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection) to main memory. In output operations, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection, and so on).

When program execution begins, three streams are connected to the program automatically. Normally, the standard input stream is connected to the keyboard and the standard output stream is connected to the screen. Operating systems often allow these streams to be redirected to other devices. A third stream, the standard error stream, is connected to the screen. Error messages are output to the standard error stream. Streams are discussed in detail in Chapter 11, C File Processing.

9.3 Formatting Output with `printf`

Precise output formatting is accomplished with `printf`. Every `printf` call contains a format control string that describes the output format. The format control string consists of conversion specifiers, flags, field widths, precisions and literal characters. Together with the percent sign (%), these form conversion specifications. Function `printf` can perform the following formatting capabilities, each of which is discussed in this chapter:

1. Rounding floating-point values to an indicated number of decimal places.
2. Aligning a column of numbers with decimal points appearing one above the other.
3. Right justification and left justification of outputs.
Chapter 9  C Formatted Input/Output

4. Inserting literal characters at precise locations in a line of output.

5. Representing floating-point numbers in exponential format.

6. Representing unsigned integers in octal and hexadecimal format. See Appendix C
for more information on octal and hexadecimal values.

7. Displaying all types of data with fixed-size field widths and precisions.

The printf function has the form

`printf( format-control-string, other-arguments );`

`format-control-string` describes the output format, and `other-arguments` (which are optional)
correspond to each conversion specification in `format-control-string`. Each conversion spec-
ification begins with a percent sign and ends with a conversion specifier. There can be
many conversion specifications in one format control string.

**Common Programming Error 9.1**

Forgetting to enclose a format-control-string in quotation marks is a syntax error.

**Good Programming Practice 9.1**

Format outputs neatly for presentation to make program outputs more readable and re-
duce user errors.

### 9.4 Printing Integers

An integer is a whole number, such as 776, 0 or −52, that contains no decimal point. In-
teger values are displayed in one of several formats. Figure 9.1 describes the integer con-
version specifiers.

<table>
<thead>
<tr>
<th>Conversion specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Display as a signed decimal integer.</td>
</tr>
<tr>
<td>i</td>
<td>Display as a signed decimal integer. [Note: The <code>i</code> and <code>d</code> specifiers are different when used with <code>scanf</code>.]</td>
</tr>
<tr>
<td>o</td>
<td>Display as an unsigned octal integer.</td>
</tr>
<tr>
<td>u</td>
<td>Display as an unsigned decimal integer.</td>
</tr>
<tr>
<td>x or X</td>
<td>Display as an unsigned hexadecimal integer. <code>x</code> causes the digits 0-9 and the letters A-F to be displayed and <code>X</code> causes the digits 0-9 and a-f to be displayed.</td>
</tr>
<tr>
<td>h or l (letter l)</td>
<td>Place before any integer conversion specifier to indicate that a short or long integer is displayed, respectively. Letters <code>h</code> and <code>l</code> are more precisely called <strong>length modifiers</strong>.</td>
</tr>
</tbody>
</table>

**Fig. 9.1**  Integer conversion specifiers.

Figure 9.2 prints an integer using each of the integer conversion specifiers. Only the
minus sign prints; plus signs are suppressed. Later in this chapter we’ll see how to force
plus signs to print. Also, the value -455, when read by %u (line 15), is converted to the unsigned value 4294966841.

```c
/* Fig 9.2: fig09_02.c */
/* Using the integer conversion specifiers */
#include <stdio.h>

int main( void )
{
    printf( "%d\n", 455 );
    printf( "%i\n", 455 ); /* i same as d in printf */
    printf( "%d\n", -455 );
    printf( "%ld\n", 2000000000L ); /* L suffix makes literal a long */
    printf( "%o\n", 455 );
    printf( "%u\n", 455 );
    printf( "%u\n", -455 );
    printf( "%x\n", 455 );
    printf( "%X\n", 455 );
    return 0; /* indicates successful termination */
} /* end main */
```

Fig. 9.2 | Using integer conversion specifiers.

Common Programming Error 9.2
Printing a negative value with a conversion specifier that expects an unsigned value.

9.5 Printing Floating-Point Numbers

A floating-point value contains a decimal point as in 33.5, 0.0 or -657.983. Floating-point values are displayed in one of several formats. Figure 9.3 describes the floating-point conversion specifiers. The conversion specifiers e and E display floating-point values in exponential notation—the computer equivalent of scientific notation used in mathematics. For example, the value 150.4582 is represented in scientific notation as

\[ 1.504582 \times 10^2 \]

and is represented in exponential notation as

\[ 1.504582E+02 \]
by the computer. This notation indicates that 1.504582 is multiplied by 10 raised to the second power (E+02). The E stands for “exponent.”

Values displayed with the conversion specifiers e, E and f show six digits of precision to the right of the decimal point by default (e.g., 1.04592); other precisions can be specified explicitly. Conversion specifier f always prints at least one digit to the left of the decimal point. Conversion specifiers e and E print lowercase e and uppercase E, respectively, preceding the exponent, and print exactly one digit to the left of the decimal point.

Conversion specifier g (or G) prints in either e (E) or f format with no trailing zeros (1.234000 is printed as 1.234). Values are printed with e (E) if, after conversion to exponential notation, the value’s exponent is less than -4, or the exponent is greater than or equal to the specified precision (six significant digits by default for g and G). Otherwise, conversion specifier f is used to print the value. Trailing zeros are not printed in the fractional part of a value output with g or G. At least one decimal digit is required for the decimal point to be output. The values 0.0000875, 8750000.0, 8.75, 87.50 and 875 are printed as 8.75e-05, 8.75e+06, 8.75, 87.5 and 875 with the conversion specifier g. The value 0.0000875 uses e notation because, when it’s converted to exponential notation, its exponent (-5) is less than -4. The value 8750000.0 uses e notation because its exponent (6) is equal to the default precision.

<table>
<thead>
<tr>
<th>Conversion specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e or E</td>
<td>Display a floating-point value in exponential notation.</td>
</tr>
<tr>
<td>f</td>
<td>Display floating-point values in fixed-point notation. [Note: In C99, you can also use f.]</td>
</tr>
<tr>
<td>g or G</td>
<td>Display a floating-point value in either the floating-point form f or the exponential form e (or E), based on the magnitude of the value.</td>
</tr>
<tr>
<td>L</td>
<td>Place before any floating-point conversion specifier to indicate that a long double floating-point value is displayed.</td>
</tr>
</tbody>
</table>

Figure 9.3 | Floating-point conversion specifiers.

The precision for conversion specifiers g and G indicates the maximum number of significant digits printed, including the digit to the left of the decimal point. The value 1234567.0 is printed as 1.23457e+06, using conversion specifier %g (remember that all floating-point conversion specifiers have a default precision of 6). There are 6 significant digits in the result. The difference between g and G is identical to the difference between e and E when the value is printed in exponential notation—lowercase g causes a lowercase e to be output, and uppercase G causes an uppercase E to be output.

Error-Prevention Tip 9.1
When outputting data, be sure that the user is aware of situations in which data may be imprecise due to formatting (e.g., rounding errors from specifying precisions).

Figure 9.4 demonstrates each of the floating-point conversion specifiers. The %E, %e and %g conversion specifiers cause the value to be rounded in the output and the conversion specifier %f does not. [Note: With some compilers, the exponent in the outputs will be shown with two digits to the right of the + sign.]
9.6 Printing Strings and Characters

The \texttt{c} and \texttt{s} conversion specifiers are used to print individual characters and strings, respectively. \textbf{Conversion specifier \texttt{c}} requires a \texttt{char} argument. \textbf{Conversion specifier \texttt{s}} requires a pointer to \texttt{char} as an argument. Conversion specifier \texttt{s} causes characters to be printed until a terminating null (‘\texttt{\\0}’) character is encountered. The program shown in Fig. 9.5 displays characters and strings with conversion specifiers \texttt{c} and \texttt{s}.

\begin{verbatim}
/* Fig 9.4: fig09_04.c */
/* Printing floating-point numbers with floating-point conversion specifiers */

#include <stdio.h>

int main( void )
{
    printf( "%e\n", 1234567.89 );
    printf( "%e\n", +1234567.89 );
    printf( "%e\n", -1234567.89 );
    printf( "%E\n", 1234567.89 );
    printf( "%f\n", 1234567.89 );
    printf( "%g\n", 1234567.89 );
    printf( "%G\n", 1234567.89 );
    return 0; /* indicates successful termination */
} /* end main */
\end{verbatim}

\begin{verbatim}
1.234568e+006
1.234568e+006
-1.234568e+006
1.234568E+006
1234567.890000
1.23457e+006
1.23457e+006
\end{verbatim}

\textbf{Fig. 9.4} | Using floating-point conversion specifiers.

9.6 Printing Strings and Characters

Common Programming Error 9.3

\textit{Using \texttt{%c} to print a string is an error. The conversion specifier \texttt{%c} expects a char argument. A string is a pointer to char (i.e., a char *).}

Common Programming Error 9.4

\textit{Using \texttt{%s} to print a char argument often causes a fatal execution-time error called an access violation. The conversion specifier \texttt{%s} expects an argument of type pointer to char.}

Common Programming Error 9.5

\textit{Using single quotes around character strings is a syntax error. Character strings must be enclosed in double quotes.}

Common Programming Error 9.6

\textit{Using double quotes around a character constant creates a pointer to a string consisting of two characters, the second of which is the terminating null.}
Chapter 9  C Formatted Input/Output

9.7 Other Conversion Specifiers

The three remaining conversion specifiers are \texttt{p}, \texttt{n} and \texttt{\%} (Fig. 9.6). The conversion specifier \texttt{\%n} stores the number of characters output so far in the current \texttt{printf}—the corresponding argument is a pointer to an integer variable in which the value is stored—nothing is printed by a \texttt{\%n}. The conversion specifier \texttt{\%} causes a percent sign to be output.

<table>
<thead>
<tr>
<th>Conversion specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{p}</td>
<td>Display a pointer value in an implementation-defined manner.</td>
</tr>
<tr>
<td>\texttt{n}</td>
<td>Store the number of characters already output in the current \texttt{printf} statement. A pointer to an integer is supplied as the corresponding argument. Nothing is displayed.</td>
</tr>
<tr>
<td>\texttt{%}</td>
<td>Display the percent character.</td>
</tr>
</tbody>
</table>

Figure 9.7’s \texttt{\%p} prints the value of \texttt{ptr} and the address of \texttt{x}; these values are identical because \texttt{ptr} is assigned the address of \texttt{x}. Next, \texttt{\%n} stores the number of characters output by the third \texttt{printf} statement (line 15) in integer variable \texttt{y}, and the value of \texttt{y} is printed. The last \texttt{printf} statement (line 21) uses \texttt{\%\%} to print the \texttt{\%} character in a character string. Every \texttt{printf} call returns a value—either the number of characters output, or a negative value if an output error occurs. [Note: This example will not execute in Microsoft Visual C++ because \texttt{\%n} has been disabled by Microsoft “for security reasons.” To execute the rest of the program, remove lines 15–16.]
9.8 Printing with Field Widths and Precision

The exact size of a field in which data is printed is specified by a field width. If the field width is larger than the data being printed, the data will normally be right justified within that field. An integer representing the field width is inserted between the percent sign (%) and the conversion specifier (e.g., %4d). Figure 9.8 prints two groups of five numbers each, right justifying those numbers that contain fewer digits than the field width. The field

```
/* Fig 9.7: fig09_07.c */
/* Using the p, n, and % conversion specifiers */
#include <stdio.h>

int main( void )
{
    int *ptr; /* define pointer to int */
    int x = 12345; /* initialize int x */
    int y; /* define int y */
    
    ptr = &x; /* assign address of x to ptr */
    printf( "The value of ptr is %p\n", ptr );
    printf( "The address of x is %p\n", &x );

    printf( "Total characters printed on this line:%n", &y );
    printf( "This line has 28 characters\n" );
    printf( "%d characters were printed\n", y );

    printf( "Printing a % in a format control string\n" );
    return 0; /* indicates successful termination */
} /* end main */
```

The value of ptr is 0012FF78
The address of x is 0012FF78
Total characters printed on this line: 38
This line has 28 characters
28 characters were printed
Printing a % in a format control string

Fig. 9.7 | Using the p, n and % conversion specifiers.
width is increased to print values wider than the field and that the minus sign for a negative value uses one character position in the field width. Field widths can be used with all conversion specifiers.

**Common Programming Error 9.8**

Not providing a sufficiently large field width to handle a value to be printed can offset other data being printed and can produce confusing outputs. Know your data!

```c
/* Fig 9.8: fig09_08.c */
/* Printing integers right-justified */
#include <stdio.h>

int main( void )
{
    printf( "%4d\n", 1 );
    printf( "%4d\n", 12 );
    printf( "%4d\n", 123 );
    printf( "%4d\n", 1234 );
    printf( "%4d\n", 12345 );

    printf( "%4d\n", -1 );
    printf( "%4d\n", -12 );
    printf( "%4d\n", -123 );
    printf( "%4d\n", -1234 );
    printf( "%4d\n", -12345 );

    return 0; /* indicates successful termination */
}
/* end main */
```

**Fig. 9.8** | Right justifying integers in a field.

Function `printf` also enables you to specify the precision with which data is printed. Precision has different meanings for different data types. When used with integer conversion specifiers, precision indicates the minimum number of digits to be printed. If the printed value contains fewer digits than the specified precision and the precision value has a leading zero or decimal point, zeros are prefixed to the printed value until the total number of digits is equivalent to the precision. If neither a zero nor a decimal point is present in the precision value, spaces are inserted instead. The default precision for integers is 1. When used with floating-point conversion specifiers `e`, `E` and `f`, the precision is the number of digits to appear after the decimal point. When used with conversion specifiers `g` and `G`, the precision is the maximum number of significant digits to be printed. When
used with conversion specifier \texttt{s}, the precision is the maximum number of characters to be written from the string. To use precision, place a decimal point (.), followed by an integer representing the precision between the percent sign and the conversion specifier. Figure 9.9 demonstrates the use of precision in format control strings. When a floating-point value is printed with a precision smaller than the original number of decimal places in the value, the value is rounded.

```c
/* Fig 9.9: fig09_09.c */
/* Using precision while printing integers, floating-point numbers, and strings */
#include <stdio.h>

int main( void )
{
    int i = 873; /* initialize int i */
    double f = 123.94536; /* initialize double f */
    char s[] = "Happy Birthday"; /* initialize char array s */

    printf( "Using precision for integers\n" );
    printf( "\t%.4d\n\t%.9d\n\n", i, i );

    printf( "Using precision for floating-point numbers\n" );
    printf( "\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f );

    printf( "Using precision for strings\n" );
    printf( "\t%.11s\n", s );

    return 0; /* indicates successful termination */
} /* end main */
```

**Fig. 9.9** | Using precisions to display information of several types.

The field width and the precision can be combined by placing the field width, followed by a decimal point, followed by a precision between the percent sign and the conversion specifier, as in the statement

```c
printf( "\%9.3f", 123.456789 );
```

which displays 123.457 with three digits to the right of the decimal point right justified in a nine-digit field.

It’s possible to specify the field width and the precision using integer expressions in the argument list following the format control string. To use this feature, insert an asterisk
(*) in place of the field width or precision (or both). The matching int argument in the argument list is evaluated and used in place of the asterisk. A field width’s value may be either positive or negative (which causes the output to be left justified in the field as described in the next section). The statement

```c
printf( "%*.*f", 7, 2, 98.736 );
```

uses 7 for the field width, 2 for the precision and outputs the value 98.74 right justified.

### 9.9 Using Flags in the printf Format Control String

Function `printf` also provides flags to supplement its output formatting capabilities. Five flags are available for use in format control strings (Fig. 9.10). To use a flag in a format control string, place the flag immediately to the right of the percent sign. Several flags may be combined in one conversion specifier.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (minus sign)</td>
<td>Left justify the output within the specified field.</td>
</tr>
<tr>
<td>+ (plus sign)</td>
<td>Display a plus sign preceding positive values and a minus sign preceding negative values.</td>
</tr>
<tr>
<td>space</td>
<td>Print a space before a positive value not printed with the + flag.</td>
</tr>
<tr>
<td>#</td>
<td>Prefix 0 to the output value when used with the octal conversion specifier o.</td>
</tr>
<tr>
<td></td>
<td>Prefix 0x or 0X to the output value when used with the hexadecimal conversion specifiers x or X.</td>
</tr>
<tr>
<td></td>
<td>Force a decimal point for a floating-point number printed with e, E, f, g or G that does not contain a fractional part. (Normally the decimal point is printed only if a digit follows it.) For g and G specifiers, trailing zeros are not eliminated.</td>
</tr>
<tr>
<td>0 (zero)</td>
<td>Pad a field with leading zeros.</td>
</tr>
</tbody>
</table>

**Fig. 9.10** | Format control string flags.

Figure 9.11 demonstrates right justification and left justification of a string, an integer, a character and a floating-point number.

```c
/* Fig 9.11: fig09_11.c */
/* Right justifying and left justifying values */
#include <stdio.h>

int main( void )
{
    printf( "%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23 );
    printf( "%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23 );
    return 0; /* indicates successful termination */
} /* end main */
```

**Fig. 9.11** | Left justifying strings in a field. (Part 1 of 2.)
9.9 Using Flags in the printf Format Control String

Figure 9.12 prints a positive number and a negative number, each with and without the + flag. The minus sign is displayed in both cases, but the plus sign is displayed only when the + flag is used.

```c
/* Fig 9.12: fig09_12.c */
/* Printing numbers with and without the + flag */
#include <stdio.h>

int main(void)
{
    printf("%d\n%d\n", 786, -786);
    printf("%+d\n%+d\n", 786, -786);
    return 0; /* indicates successful termination */
} /* end main */
```

786
-786
+786
-786

Figure 9.12 | Printing positive and negative numbers with and without the + flag.

Figure 9.13 prefixes a space to the positive number with the space flag. This is useful for aligning positive and negative numbers with the same number of digits. The value -547 is not preceded by a space in the output because of its minus sign.

```c
/* Fig 9.13: fig09_13.c */
/* Printing a space before signed values not preceded by + or - */
#include <stdio.h>

int main(void)
{
    printf("% d\n%d\n", 547, -547);
    return 0; /* indicates successful termination */
} /* end main */
```

547
-547

Figure 9.13 | Using the space flag.

Figure 9.14 uses the # flag to prefix 0 to the octal value and 0x and 0X to the hexadecimal values, and to force the decimal point on a value printed with g.
Chapter 9  C Formatted Input/Output

Figure 9.15 combines the + flag and the 0 (zero) flag to print 452 in a 9-space field with a + sign and leading zeros, then prints 452 again using only the 0 flag and a 9-space field.

9.10 Printing Literals and Escape Sequences

Most literal characters to be printed in a printf statement can simply be included in the format control string. However, there are several “problem” characters, such as the quotation mark (") that delimits the format control string itself. Various control characters, such
as newline and tab, must be represented by escape sequences. An escape sequence is represented by a backslash (\), followed by a particular escape character. Figure 9.16 lists the escape sequences and the actions they cause.

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'</code> (single quote)</td>
<td>Output the single quote (') character.</td>
</tr>
<tr>
<td><code>&quot;</code> (double quote)</td>
<td>Output the double quote (&quot;) character.</td>
</tr>
<tr>
<td><code>?</code> (question mark)</td>
<td>Output the question mark (?) character.</td>
</tr>
<tr>
<td><code>\</code> (backslash)</td>
<td>Output the backslash () character.</td>
</tr>
<tr>
<td><code>a</code> (alert or bell)</td>
<td>Cause an audible (bell) or visual alert.</td>
</tr>
<tr>
<td><code>b</code> (backspace)</td>
<td>Move the cursor back one position on the current line.</td>
</tr>
<tr>
<td><code>f</code> (new page or form feed)</td>
<td>Move the cursor to the start of the next logical page.</td>
</tr>
<tr>
<td><code>n</code> (newline)</td>
<td>Move the cursor to the beginning of the next line.</td>
</tr>
<tr>
<td><code>r</code> (carriage return)</td>
<td>Move the cursor to the beginning of the current line.</td>
</tr>
<tr>
<td><code>t</code> (horizontal tab)</td>
<td>Move the cursor to the next horizontal tab position.</td>
</tr>
<tr>
<td><code>v</code> (vertical tab)</td>
<td>Move the cursor to the next vertical tab position.</td>
</tr>
</tbody>
</table>

**Fig. 9.16** | Escape sequences.

**9.11 Reading Formatted Input with scanf**

Precise input formatting can be accomplished with scanf. Every scanf statement contains a format control string that describes the format of the data to be input. The format control string consists of conversion specifiers and literal characters. Function scanf has the following input formatting capabilities:

1. Inputting all types of data.
2. Inputting specific characters from an input stream.
3. Skipping specific characters in the input stream.

Function scanf is written in the following form:

```c
scanf( format-control-string, other-arguments );
```

*format-control-string* describes the formats of the input, and *other-arguments* are pointers to variables in which the input will be stored.

**Good Programming Practice 9.2**

When inputting data, prompt the user for one data item or a few data items at a time. Avoid asking the user to enter many data items in response to a single prompt.
**Good Programming Practice 9.3**

*Always consider what the user and your program will do when (not if) incorrect data is entered—for example, a value for an integer that is nonsensical in a program’s context, or a string with missing punctuation or spaces.*

Figure 9.17 summarizes the conversion specifiers used to input all types of data. The remainder of this section provides programs that demonstrate reading data with the various `scanf` conversion specifiers.

<table>
<thead>
<tr>
<th>Conversion specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Integers</strong></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>Read an optionally signed decimal integer. The corresponding argument is a pointer to an <code>int</code>.</td>
</tr>
<tr>
<td>i</td>
<td>Read an optionally signed decimal, octal or hexadecimal integer. The corresponding argument is a pointer to an <code>int</code>.</td>
</tr>
<tr>
<td>o</td>
<td>Read an octal integer. The corresponding argument is a pointer to an unsigned <code>int</code>.</td>
</tr>
<tr>
<td>u</td>
<td>Read an unsigned decimal integer. The corresponding argument is a pointer to an unsigned <code>int</code>.</td>
</tr>
<tr>
<td>x or X</td>
<td>Read a hexadecimal integer. The corresponding argument is a pointer to an unsigned <code>int</code>.</td>
</tr>
<tr>
<td>h or l</td>
<td>Place before any of the integer conversion specifiers to indicate that a short or long integer is to be input.</td>
</tr>
<tr>
<td><strong>Floating-point numbers</strong></td>
<td></td>
</tr>
<tr>
<td>e, E, f, g or G</td>
<td>Read a floating-point value. The corresponding argument is a pointer to a floating-point variable.</td>
</tr>
<tr>
<td>l or L</td>
<td>Place before any of the floating-point conversion specifiers to indicate that a double or long double value is to be input. The corresponding argument is a pointer to a double or long double variable.</td>
</tr>
<tr>
<td><strong>Characters and strings</strong></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>Read a character. The corresponding argument is a pointer to a char; no null (<code>\0</code>) is added.</td>
</tr>
<tr>
<td>s</td>
<td>Read a string. The corresponding argument is a pointer to an array of type <code>char</code> that is large enough to hold the string and a terminating null (<code>\0</code>) character—which is automatically added.</td>
</tr>
<tr>
<td><strong>Scan set</strong></td>
<td></td>
</tr>
<tr>
<td>[scan characters]</td>
<td>Scan a string for a set of characters that are stored in an array.</td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>Read an address of the same form produced when an address is output with <code>%p</code> in a <code>printf</code> statement.</td>
</tr>
<tr>
<td>n</td>
<td>Store the number of characters input so far in this call to <code>scanf</code>. The corresponding argument is a pointer to an <code>int</code>.</td>
</tr>
<tr>
<td>%</td>
<td>Skip a percent sign (%) in the input.</td>
</tr>
</tbody>
</table>
Figure 9.18 reads integers with the various integer conversion specifiers and displays the integers as decimal numbers. Conversion specifier %i is capable of inputting decimal, octal and hexadecimal integers.

```c
/* Fig 9.18: fig09_18.c */
/* Reading integers */
#include <stdio.h>

int main( void )
{
    int a;
    int b;
    int c;
    int d;
    int e;
    int f;
    int g;

    printf( "Enter seven integers: " );
    scanf( "%d%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g );

    printf( "The input displayed as decimal integers is:\n" );
    printf( "%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g );
    return 0; /* indicates successful termination */
} /* end main */
```

Enter seven integers: -70 -70 070 0x70 70 70 70
The input displayed as decimal integers is:
-70 -70 56 112 56 70 112

Fig. 9.18 | Reading input with integer conversion specifiers.

When inputting floating-point numbers, any of the floating-point conversion specifiers e, E, f, g or G can be used. Figure 9.19 reads three floating-point numbers, one with each of the three types of floating conversion specifiers, and displays all three numbers with conversion specifier f. The program output confirms the fact that floating-point values are imprecise—this is highlighted by the third value printed.

```c
/* Fig 9.19: fig09_19.c */
/* Reading floating-point numbers */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    double a;
    double b;
    double c;
```

Fig. 9.19 | Reading input with floating-point conversion specifiers. (Part 1 of 2.)
Chapter 9  C Formatted Input/Output

Characters and strings are input using the conversion specifiers \texttt{c} and \texttt{s}, respectively. Figure 9.20 prompts the user to enter a string. The program inputs the first character of the string with \texttt{%c} and stores it in the character variable \texttt{x}, then inputs the remainder of the string with \texttt{%s} and stores it in character array \texttt{y}.

```c
int main( void )
{
    char x;
    char y[ 9 ];
    printf( "Enter a string: ");
    scanf( "\%c\%s", &x, y );
    printf( "The input was: \n" );
    printf( "the character \"%c\"", x );
    printf( "and the string \"%s\"\n", y );
    return 0; /* indicates successful termination */
}
```

Enter a string: Sunday
The input was:
the character "S" and the string "unday"

A sequence of characters can be input using a \textit{scan set}. A scan set is a set of characters enclosed in square brackets, [], and preceded by a percent sign in the format control string. A scan set scans the characters in the input stream, looking only for those characters that match characters contained in the scan set. Each time a character is matched, it's
stored in the scan set’s corresponding argument—a pointer to a character array. The scan set stops inputting characters when a character that is not contained in the scan set is encountered. If the first character in the input stream does not match a character in the scan set, only the null character is stored in the array. Figure 9.21 uses the scan set \[aeiou\] to scan the input stream for vowels. Notice that the first seven letters of the input are read. The eighth letter (h) is not in the scan set and therefore the scanning is terminated.

```c
/* Fig 9.21: fig09_21.c */
/*@ Using a scan set */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    char z[ 9 ]; /* define array z */
    printf( "Enter string: " );
    scanf("%[aeiou]", z ); /* search for set of characters */
    printf( "The input was \"%s\"\n", z );
    return 0; /* indicates successful termination */
} /* end main */
```

Enter string: ooeeooahah
The input was "ooeeooa"

Fig. 9.21 | Using a scan set.

The scan set can also be used to scan for characters not contained in the scan set by using an \textit{inverted scan set}. To create an inverted scan set, place a \texttt{caret (^)} in the square brackets before the scan characters. This causes characters not appearing in the scan set to be stored. When a character contained in the inverted scan set is encountered, input terminates. Figure 9.22 uses the inverted scan set \[^aeiou\] to search for consonants—more properly to search for “nonvowels.”

```c
/* Fig 9.22: fig09_22.c */
/*@ Using an inverted scan set */
#include <stdio.h>

int main( void )
{
    char z[ 9 ];
    printf("Enter a string: ");
    scanf("%[^aeiou]", z ); /* inverted scan set */
    printf("The input was \"%s\"\n", z );
    return 0; /* indicates successful termination */
} /* end main */
```

Fig. 9.22 | Using an inverted scan set. (Part 1 of 2.)
A field width can be used in a scanf conversion specifier to read a specific number of characters from the input stream. Figure 9.23 inputs a series of consecutive digits as a two-digit integer and an integer consisting of the remaining digits in the input stream.

Often it’s necessary to skip certain characters in the input stream. For example, a date could be entered as

```
11-10-1999
```

Each number in the date needs to be stored, but the dashes that separate the numbers can be discarded. To eliminate unnecessary characters, include them in the format control string of scanf (white-space characters—such as space, newline and tab—skip all leading white-space). For example, to skip the dashes in the input, use the statement

```
scanf("%d-%d-%d", &month, &day, &year);
```

Although, this scanf does eliminate the dashes in the preceding input, it’s possible that the date could be entered as

```
10/11/1999
```

In this case, the preceding scanf would not eliminate the unnecessary characters. For this reason, scanf provides the assignment suppression character *. The assignment suppression character enables scanf to read any type of data from the input and discard it without assigning it to a variable. Figure 9.24 uses the assignment suppression character in the %c conversion specifier to indicate that a character appearing in the input stream should be
read and discarded. Only the month, day and year are stored. The values of the variables are printed to demonstrate that they are in fact input correctly. The argument lists for each scanf call do not contain variables for the conversion specifiers that use the assignment suppression character. The corresponding characters are simply discarded.

```c
/* Fig 9.24: fig09_24.c */
/* Reading and discarding characters from the input stream */
#include <stdio.h>

int main( void )
{
    int month1;
    int day1;
    int year1;
    int month2;
    int day2;
    int year2;

    printf( "Enter a date in the form mm-dd-yyyy: ");
    scanf( "%d%*c%d%*c%d", &month1, &day1, &year1 );
    printf( "month = %d  day = %d  year = %d
\n", month1, day1, year1 );

    printf( "Enter a date in the form mm/dd/yyyy: ");
    scanf( "%d%*c%d%*c%d", &month2, &day2, &year2 );
    printf( "month = %d  day = %d  year = %d
", month2, day2, year2 );
    return 0; /* indicates successful termination */
} /* end main */
```

Enter a date in the form mm-dd-yyyy: 11-18-2003
month = 11  day = 18  year = 2003

Enter a date in the form mm/dd/yyyy: 11/18/2003
month = 11  day = 18  year = 2003

Fig. 9.24 | Reading and discarding characters from the input stream.

**Summary**

**Section 9.2 Streams**
- All input and output is dealt with in streams—sequences of characters organized into lines. Each line consists of zero or more characters and ends with a newline character.
- Normally, the standard input stream is connected to the keyboard, and the standard output stream is connected to the computer screen.
- Operating systems often allow the standard input and standard output streams to be redirected to other devices.

**Section 9.3 Formatting Output with printf**
- A format control string describes the formats in which the output values appear. The format control string consists of conversion specifiers, flags, field widths, precisions and literal characters.
Section 9.4 Printing Integers
• Integers are printed with the following conversion specifiers: d or i for optionally signed integers, o for unsigned integers in octal form, u for unsigned integers in decimal form and x or X for unsigned integers in hexadecimal form. The modifier h or l is prefixed to the preceding conversion specifiers to indicate a short or long integer, respectively.

Section 9.5 Printing Floating-Point Numbers
• Floating-point values are printed with the following conversion specifiers: e or E for exponential notation, f for regular floating-point notation, and g or G for either e (or E) notation or f notation. When the g (or G) conversion specifier is indicated, the e (or E) conversion specifier is used if the value's exponent is less than -4 or greater than or equal to the precision with which the value is printed.
• The precision for the g and G conversion specifiers indicates the maximum number of significant digits printed.

Section 9.6 Printing Strings and Characters
• The conversion specifier c prints a character.
• The conversion specifier s prints a string of characters ending in the null character.

Section 9.7 Other Conversion Specifiers
• The conversion specifier p displays an address in an implementation-defined manner (on many systems, hexadecimal notation is used).
• The conversion specifier n stores the number of characters already output in the current printf statement. The corresponding argument is a pointer to an int.
• The conversion specifier %% causes a literal % to be output.

Section 9.8 Printing with Field Widths and Precision
• If the field width is larger than the object being printed, the object is right justified by default.
• Field widths can be used with all conversion specifiers.
• Precision used with integer conversion specifiers indicates the minimum number of digits printed. Zeros are prefixed to the printed value until the number of digits is equivalent to the precision.
• Precision used with floating-point conversion specifiers e, E and f indicates the number of digits that appear after the decimal point. Precision used with floating-point conversion specifiers g and G indicates the number of significant digits to appear.
• Precision used with conversion specifier s indicates the number of characters to be printed.
• The field width and the precision can be combined by placing the field width, followed by a decimal point, followed by the precision between the percent sign and the conversion specifier.
• It's possible to specify the field width and the precision through integer expressions in the argument list following the format control string. To do so, use an asterisk (*) for the field width or precision. The matching argument in the argument list is used in place of the asterisk. The value of the argument can be negative for the field width but must be positive for the precision.

Section 9.9 Using Flags in the printf Format Control String
• The - flag left justifies its argument in a field.
• The + flag prints a plus sign for positive values and a minus sign for negative values. The space flag prints a space preceding a positive value not displayed with the + flag.
• The # flag prefixes 0 to octal values and 0x or 0X to hexadecimal values, and forces the decimal point to be printed for floating-point values printed with e, E, f, g or G.
• The 0 flag prints leading zeros for a value that does not occupy its entire field width.
Section 9.10 Printing Literals and Escape Sequences

- Most literal characters to be printed in a printf statement can simply be included in the format control string. However, there are several “problem” characters, such as the quotation mark ("), that delimits the format control string itself. Various control characters, such as newline and tab, must be represented by escape sequences. An escape sequence is represented by a backslash (\\), followed by a particular escape character.

Section 9.11 Formatting Input with scanf

- Precise input formatting is accomplished with the scanf library function.
- Integers are input with scanf with the conversion specifiers d and i for optionally signed integers and o, u, x or X for unsigned integers. The modifiers h and l are placed before an integer conversion specifier to input a short or long integer, respectively.
- Floating-point values are input with scanf with the conversion specifiers e, E, f, g or G. The modifiers l and L are placed before any of the floating-point conversion specifiers to indicate that the input value is a double or long double value, respectively.
- Characters are input with scanf with the conversion specifier c.
- Strings are input with scanf with the conversion specifier s.
- A scan set scans the characters in the input, looking only for those characters that match characters contained in the scan set. When a character is matched, it’s stored in a character array. The scan set stops inputting characters when a character not contained in the scan set is encountered.
- To create an inverted scan set, place a caret (^) in the square brackets before the scan characters. This causes characters input with scanf and not appearing in the scan set to be stored until a character contained in the inverted scan set is encountered.
- Address values are input with scanf with the conversion specifier p.
- Conversion specifier n stores the number of characters input previously in the current scanf. The corresponding argument is a pointer to int.
- The conversion specifier % with scanf matches a single % character in the input.
- The assignment suppression character reads data from the input stream and discards the data.
- A field width is used in scanf to read a specific number of characters from the input stream.

Terminology

" (quotation mark) 368
* assignment suppression character 374
# flag 367
% character in a conversion specifier 357
%c conversion specifier 362
%c conversion specifier 361
%c conversion specifier 359
%c conversion specifier 360
%c (or %c) conversion specifier 360
%c conversion specifier 371
%p conversion specifier 362
%p conversion specifier 361, 372
%p conversion specifier 359
+ flag 367
0 (zero) flag 368
caret (\^) 373
conversion specification 357
conversion specifier 357
exponential notation 359
field width 357
flag 357
format control string 357
integer conversion specifier 358
inverted scan set 373
left justification 357
length modifier 358
literal modifier 357
precision 357
printf function 357
right justification 357
rounding 357
scanf function 357
Chapter 9  C Formatted Input/Output

Self-Review Exercises

9.1 Fill in the blanks in each of the following:
   a) All input and output is dealt with in the form of _______.
   b) The _______ stream is normally connected to the keyboard.
   c) The _______ stream is normally connected to the computer screen.
   d) Precise output formatting is accomplished with the _______ function.
   e) The format control string may contain _______.
   f) The conversion specifier _______ or _______ may be used to output a signed decimal integer.
   g) The conversion specifiers _______, _______, and _______ are used to display unsigned integers in octal, decimal and hexadecimal form, respectively.
   h) The modifiers _______. and _______. are placed before the integer conversion specifiers to indicate that _______ or _______ integer values are to be displayed.
   i) The conversion specifier _______ is used to display a floating-point value in exponential notation.
   j) The modifier _______ is placed before any floating-point conversion specifier to indicate that a _______ _______ value is to be displayed.
   k) The conversion specifiers _______, _______, and _______ are displayed with _______ digits of precision to the right of the decimal point if no precision is specified.
   l) The conversion specifiers _______ and _______ are used to print strings and characters, respectively.
   m) All strings end in the _______ character.
   n) The field width and precision in a printf conversion specifier can be controlled with integer expressions by substituting _______ for the field width or for the precision and placing an integer expression in the corresponding argument of the argument list.
   o) The _______. flag causes output to be left justified in a field.
   p) The _______. flag causes values to be displayed with either a plus sign or a minus sign.
   q) Precise input formatting is accomplished with the _______. function.
   r) A(n) _______. is used to scan a string for specific characters and store the characters in an array.
   s) The conversion specifier _______. can be used to input optionally signed octal, decimal and hexadecimal integers.
   t) The conversion specifiers _______. can be used to input a _______ value.
   u) The _______. is used to read data from the input stream and discard it without assigning it to a variable.
   v) A(n) _______. can be used in a scanf conversion specifier to indicate that a specific number of characters or digits should be read from the input stream.

9.2 Find the error in each of the following and explain how the error can be corrected.
   a) The following statement should print the character 'c'.
      printf( "%s\n", 'c' );
   b) The following statement should print 9.375%.
      printf( "%3f\%", 9.375 );
   c) The following statement should print the first character of the string "Monday".
      printf( "%c\n", "Monday" );
d) `printf("A string in quotes");`

c) `printf("%d%d", 12, 20);`

e) `printf("%d%d", 12, 20);`

f) `printf("%c", "x");`

g) `printf("%s\n", 'Richard');`

9.3 Write a statement for each of the following:

a) Print 1234 right justified in a 10-digit field.

b) Print 123.456789 in exponential notation with a sign (+ or -) and 3 digits of precision.

c) Read a double value into variable number.

d) Print 100 in octal form preceded by 0.

e) Read a string into character array string.

f) Read characters into array n until a nondigit character is encountered.

g) Use integer variables x and y to specify the field width and precision used to display the double value 87.4573.

h) Read a value of the form 3.5%. Store the percentage in float variable percent and eliminate the % from the input stream. Do not use the assignment suppression character.

i) Print 3.333333 as a long double value with a sign (+ or -) in a field of 20 characters with a precision of 3.

Answers to Self-Review Exercises

9.1 a) Streams. b) Standard input. c) Standard output. d) `printf`. e) Conversion specifiers, flags, field widths, precisions, literal characters. f) d, i. g) o, u, x (or X). h) h, l. i) e (or E). j) L. k) 6. l) s, c, m) NULL ('\0'). n) asterisk (*). o) - (minus). p) + (plus). q) `scanf`. r) Scan set. s) i, t) 1e, 1E, 1f, 1g or 1G. u) Assignment suppression character (*).

9.2 a) Error: Conversion specifier s expects an argument of type pointer to char.

Correction: To print the character 'c', use the conversion specifier %c or change 'c' to "c".

b) Error: Trying to print the literal character % without using the conversion specifier %%. 

Correction: Use % to print a literal character.

c) Error: Conversion specifier c expects an argument of type char.

Correction: To print the first character of "Monday" use the conversion specifier %s.

d) Error: Trying to print the literal character " without using the \" escape sequence.

Correction: Replace each quote in the inner set of quotes with \\.

e) Error: The format control string is not enclosed in double quotes.

Correction: Enclose %d%d in double quotes.

f) Error: The character x is enclosed in double quotes.

Correction: Character constants to be printed with %c must be enclosed in single quotes.

g) Error: The string to be printed is enclosed in single quotes.

Correction: Use double quotes instead of single quotes to represent a string.

9.3 a) `printf("%10d\n", 1234);`

b) `printf("%+.3e\n", 123.456789);`

c) `scanf("%lf", &number);`

 d) `printf("%#o\n", 100);`

e) `scanf("%s", string);`

f) `scanf("%s[0123456789]", n);`

g) `printf("%+.2f\n", x, y, 87.4573);`

h) `scanf("%f\", &percent);`

i) `printf("%+20.3Lf\n", 3.333333);`
Exercises

9.4 Write a printf or scanf statement for each of the following:
   a) Print unsigned integer 40000 left justified in a 15-digit field with 8 digits.
   b) Read a hexadecimal value into variable hex.
   c) Print 200 with and without a sign.
   d) Print 100 in hexadecimal form preceded by 0x.
   e) Read characters into array s until the letter p is encountered.
   f) Print 1.234 in a 9-digit field with preceding zeros.
   g) Read a time of the form hh:mm:ss, storing the parts of the time in the integer variables hour, minute and second. Skip the colons (:) in the input stream. Use the assignment suppression character.
   h) Read a string of the form "characters" from the standard input. Store the string in character array s. Eliminate the quotation marks from the input stream.
   i) Read a time of the form hh:mm:ss, storing the parts of the time in the integer variables hour, minute and second. Skip the colons (:) in the input stream. Do not use the assignment suppression character.

9.5 Show what each of the following statements prints. If a statement is incorrect, indicate why.
   a) printf("%-10d\n", 10000);
   b) printf("%c\n", "This is a string");
   c) printf("%.*f\n", 8, 3, 1024.987654);
   d) printf("%#o\n%#X\n%#e\n", 17, 17, 1008.83689);
   e) printf("% ld\n%+ld\n", 1000000, 1000000);
   f) printf("%10.2E\n", 444.93738);
   g) printf("%10.2g\n", 444.93738);
   h) printf("%d\n", 10.987);

9.6 Find the error(s) in each of the following program segments. Explain how each error can be corrected.
   a) printf("%s\n", 'Happy Birthday' );
   b) printf("%c\n", 'Hello' );
   c) printf("%c\n", "This is a string" );
   d) The following statement should print "Bon Voyage":
      printf(""%s"", "Bon Voyage" );
   e) char day[] = "Sunday";
      printf("%s\n", day[ 3 ] );
   f) printf( 'Enter your name: ' );
   g) printf( %f, 123.456 );
   h) The following statement should print the characters 'O' and 'K':
      printf("%s%s\n", 'O', 'K' );
   i) char s[ 10 ];
      scanf("%c", s[ 7 ] );

9.7 (Differences Between %d and %i) Write a program to test the difference between the %d and %i conversion specifiers when used in scanf statements. Use the statements
   scanf("%%id", &x, &y );
   printf("%%d %%d", x, y );

to input and print the values. Test the program with the following sets of input data:

```
10    10
-10   -10
010   010
0x10  0x10
```
9.8  (Displaying Values and Their Number of Characters) Write a program that loads 10-element array `number` with random integers from 1 to 1000. For each value, print the value and a running total of the number of characters printed. Use the `%n` conversion specifier to determine the number of characters output for each value. Print the total number of characters output for all values up to and including the current value each time the current value is printed.

9.9  (Printing Pointer Values as Integers) Write a program that prints pointer values using all the integer conversion specifiers and the `%p` conversion specifier. Which ones print strange values? Which ones cause errors? In which format does the `%p` conversion specifier display the address on your system?

9.10 (Printing Numbers in Various Field Widths) Write a program to test the results of printing the integer value 12345 and the floating-point value 1.2345 in various size fields. What happens when the values are printed in fields containing fewer digits than the values?

9.11 (Rounding Floating-Point Numbers) Write a program that prints the value 100.453627 rounded to the nearest digit, tenth, hundredth, thousandth and ten-thousandth.

9.12 (Displaying a String in a Field) Write a program that inputs a string from the keyboard and determines the length of the string. Print the string using twice the length as the field width.

9.13 (Temperature Conversions) Write a program that converts integer Fahrenheit temperatures from 0 to 212 degrees to floating-point Celsius temperatures with 3 digits of precision. Perform the calculation using the formula

\[ \text{celsius} = \frac{5.0}{9.0} \times (\text{fahrenheit} - 32) \]

The output should be printed in two right-justified columns of 10 characters each, and the Celsius temperatures should be preceded by a sign for both positive and negative values.

9.14 (Escape Sequences) Write a program to test all the escape sequences in Figure 9.16. For the escape sequences that move the cursor, print a character before and after printing the escape sequence so it’s clear where the cursor has moved.

9.15 (Printing a Question Mark) Write a program that determines whether `?` can be printed as part of a `printf` format control string as a literal character rather than using the `\?` escape sequence.

9.16 (Reading an Integer with Each `scanf` Conversion Specifier) Write a program that inputs the value 437 using each of the `scanf` integer conversion specifiers. Print each input value using all the integer conversion specifiers.

9.17 (Outputting a Number with the Floating-Point Conversion Specifiers) Write a program that uses each of the conversion specifiers `e`, `f` and `g` to input the value 1.2345. Print the values of each variable to prove that each conversion specifier can be used to input this same value.

9.18 (Reading Strings in Quotes) In some programming languages, strings are entered surrounded by either single or double quotation marks. Write a program that reads the three strings `suzy`, "suzy" and ‘suzy’. Are the single and double quotes ignored by C or read as part of the string?

9.19 (Printing a Question Mark as a Character Constant) Write a program that determines whether `?` can be printed as the character constant ‘?’ rather than the character constant escape sequence ‘\?’ using conversion specifier `%c` in the format control string of a `printf` statement.

9.20 (Using `%g` with Various Precisions) Write a program that uses the conversion specifier `g` to output the value 9876.12345. Print the value with precisions ranging from 1 to 9.
Objectives

In this chapter, you’ll learn:

- To create and use structures, unions and enumerations.
- To pass structures to functions by value and by reference.
- To manipulate data with the bitwise operators.
- To create bit fields for storing data compactly.
10.1 Introduction

Structures—sometimes referred to as aggregates—are collections of related variables under one name. Structures may contain variables of many different data types—in contrast to arrays that contain only elements of the same data type. Structures are commonly used to define records to be stored in files (see Chapter 11, C File Processing). Pointers and structures facilitate the formation of more complex data structures such as linked lists, queues, stacks and trees (see Chapter 12, C Data Structures).

10.2 Structure Definitions

Structures are derived data types—they are constructed using objects of other types. Consider the following structure definition:

```
struct card {
    char *face;
    char *suit;
};
```

Keyword `struct` introduces the structure definition. The identifier `card` is the structure tag, which names the structure definition and is used with the keyword `struct` to declare variables of the structure type. In this example, the structure type is `struct card`. Variables declared within the braces of the structure definition are the structure’s members. Members of the same structure type must have unique names, but two different structure types may contain members of the same name without conflict (we’ll soon see why). Each structure definition must end with a semicolon.

The definition of `struct card` contains members `face` and `suit` of type `char *`. Structure members can be variables of the primitive data types (e.g., `int`, `float`, etc.), or aggregates, such as arrays and other structures. As we saw in Chapter 6, each element of an array must be of the same type. Structure members, however, can be of many types. For example, the following `struct` contains character array members for an employee’s first and last names, an `int` member for the employee’s age, a `char` member that would contain ‘M’ or ‘F’ for the employee’s gender and a `double` member for the employee’s hourly salary:
Self-Referential Structures
A structure cannot contain an instance of itself. For example, a variable of type `struct employee` cannot be declared in the definition for `struct employee`. A pointer to `struct employee`, however, may be included. For example,

```c
struct employee2 {
    char firstName[ 20 ];
    char lastName[ 20 ];
    int age;
    char gender;
    double hourlySalary;
    struct employee2 person; /* ERROR */
    struct employee2 *ePtr; /* pointer */
};
```

`struct employee2` contains an instance of itself (`person`), which is an error. Because `ePtr` is a pointer (to type `struct employee2`), it is permitted in the definition. A structure containing a member that is a pointer to the same structure type is referred to as a **self-referential structure**. Self-referential structures are used in Chapter 12 to build linked data structures.

Defining Variables of Structure Types
Structure definitions do not reserve any space in memory; rather, each definition creates a new data type that is used to define variables. Structure variables are defined like variables of other types. The definition

```c
struct card aCard, deck[ 52 ], *cardPtr;
```

declares `aCard` to be a variable of type `struct card`, declares `deck` to be an array with 52 elements of type `struct card` and declares `cardPtr` to be a pointer to `struct card`. Variables of a given structure type may also be declared by placing a comma-separated list of the variable names between the closing brace of the structure definition and the semicolon that ends the structure definition. For example, the preceding definition could have been incorporated into the `struct card` structure definition as follows:

```c
struct card {
    char *face;
    char *suit;
} aCard, deck[ 52 ], *cardPtr;
```

Structure Tag Names
The structure tag name is optional. If a structure definition does not contain a structure tag name, variables of the structure type may be declared only in the structure definition—not in a separate declaration.
10.2 Structure Definitions

**Operations That Can Be Performed on Structures**

The only valid operations that may be performed on structures are the following: assigning structure variables to structure variables of the same type, taking the address (&) of a structure variable, accessing the members of a structure variable (see Section 10.4) and using the `sizeof` operator to determine the size of a structure variable.

Structures may not be compared using operators `==` and `!=`, because structure members are not necessarily stored in consecutive bytes of memory. Sometimes there are “holes” in a structure, because computers may store specific data types only on certain memory boundaries such as half word, word or double word boundaries. A word is a standard memory unit used to store data in a computer—usually 2 bytes or 4 bytes. Consider the following structure definition, in which `sample1` and `sample2` of type `struct example` are declared:

```c
struct example {
    char c;
    int i;
} sample1, sample2;
```

A computer with 2-byte words may require that each member of `struct example` be aligned on a word boundary, i.e., at the beginning of a word (this is machine dependent). Figure 10.1 shows a sample storage alignment for a variable of type `struct example` that has been assigned the character 'a' and the integer 97 (the bit representations of the values are shown). If the members are stored beginning at word boundaries, there is a 1-byte hole (byte 1 in the figure) in the storage for variables of type `struct example`. The value in the 1-byte hole is undefined. Even if the member values of `sample1` and `sample2` are in fact equal, the structures are not necessarily equal, because the undefined 1-byte holes are not likely to contain identical values.

**Common Programming Error 10.2**

Assigning a structure of one type to a structure of a different type is a compilation error.

**Good Programming Practice 10.1**

Always provide a structure tag name when creating a structure type. The structure tag name is convenient for declaring new variables of the structure type later in the program.

**Good Programming Practice 10.2**

Choosing a meaningful structure tag name helps make a program self-documenting.

**Fig. 10.1** | Possible storage alignment for a variable of type `struct example` showing an undefined area in memory.
10.3 Initializing Structures

Structures can be initialized using initializer lists as with arrays. To initialize a structure, follow the variable name in the definition with an equals sign and a brace-enclosed, comma-separated list of initializers. For example, the declaration

```
struct card aCard = { "Three", "Hearts" };
```

creates variable aCard to be of type struct card (as defined in Section 10.2) and initializes member face to "Three" and member suit to "Hearts". If there are fewer initializers in the list than members in the structure, the remaining members are automatically initialized to 0 (or NULL if the member is a pointer). Structure variables defined outside a function definition (i.e., externally) are initialized to 0 or NULL if they are not explicitly initialized in the external definition. Structure variables may also be initialized in assignment statements by assigning a structure variable of the same type, or by assigning values to the individual members of the structure.

10.4 Accessing Structure Members

Two operators are used to access members of structures: the structure member operator (.)—also called the dot operator—and the structure pointer operator (->)—also called the arrow operator. The structure member operator accesses a structure member via the structure variable name. For example, to print member suit of structure variable aCard defined in Section 10.3, use the statement

```
printf( "%s", aCard.suit ); /* displays Hearts */
```

The structure pointer operator—consisting of a minus (-) sign and a greater than (> ) sign with no intervening spaces—accesses a structure member via a pointer to the structure. Assume that the pointer cardPtr has been declared to point to struct card and that the address of structure aCard has been assigned to cardPtr. To print member suit of structure aCard with pointer cardPtr, use the statement

```
printf( "%s", cardPtr->suit ); /* displays Hearts */
```

The expression cardPtr->suit is equivalent to (*cardPtr).suit, which dereferences the pointer and accesses the member suit using the structure member operator. The parentheses are needed here because the structure member operator (.) has a higher precedence than the pointer dereferencing operator (*). The structure pointer operator and structure member operator, along with parentheses (for calling functions) and brackets ([ ]) used for array subscripting, have the highest operator precedence and associate from left to right.

Good Programming Practice 10.3

Do not put spaces around the -> and . operators. Omitting spaces helps emphasize that the expressions the operators are contained in are essentially single variable names.
The program of Fig. 10.2 demonstrates the use of the structure member and structure pointer operators. Using the structure member operator, the members of structure aCard are assigned the values "Ace" and "Spades", respectively (lines 18 and 19). Pointer cardPtr is assigned the address of structure aCard (line 21). Function printf prints the members of structure variable aCard using the structure member operator with variable name aCard, the structure pointer operator with pointer cardPtr and the structure member operator with dereferenced pointer cardPtr (lines 23 through 25).

```c
/* Fig. 10.2: fig10_02.c
Using the structure member and structure pointer operators */
#include <stdio.h>

/* card structure definition */
struct card {
    char *face; /* define pointer face */
    char *suit; /* define pointer suit */
}; /* end structure card */

int main( void )
{
    struct card aCard; /* define one struct card variable */
    struct card *cardPtr; /* define a pointer to a struct card */

    /* place strings into aCard */
    aCard.face = "Ace";
    aCard.suit = "Spades";

    cardPtr = &aCard; /* assign address of aCard to cardPtr */

    printf( "%s\n%s\n%s\n", aCard.face, " of ", aCard.suit,
            cardPtr->face, " of ", cardPtr->suit,
            ( *cardPtr ).face, " of ", ( *cardPtr ).suit );
    return 0; /* indicates successful termination */
} /* end main */
```

Fig. 10.2 | Structure member operator and structure pointer operator. (Part 1 of 2.)
10.5 Using Structures with Functions

Structures may be passed to functions by passing individual structure members, by passing an entire structure or by passing a pointer to a structure. When structures or individual structure members are passed to a function, they are passed by value. Therefore, the members of a caller’s structure cannot be modified by the called function. To pass a structure by reference, pass the address of the structure variable. Arrays of structures—like all other arrays—are automatically passed by reference.

In Chapter 6, we stated that an array could be passed by value by using a structure. To pass an array by value, create a structure with the array as a member. Structures are passed by value, so the array is passed by value.

Common Programming Error 10.6
Assuming that structures, like arrays, are automatically passed by reference and trying to modify the caller’s structure values in the called function is a logic error.

Performance Tip 10.1
Passing structures by reference is more efficient than passing structures by value (which requires the entire structure to be copied).

10.6 typedef

The keyword typedef provides a mechanism for creating synonyms (or aliases) for previously defined data types. Names for structure types are often defined with typedef to create shorter type names. For example, the statement

```c
typedef struct card Card;
```

defines the new type name Card as a synonym for type struct card. C programmers often use typedef to define a structure type, so a structure tag is not required. For example, the following definition

```c
typedef struct {
    char *face;
    char *suit;
} Card;
```

creates the structure type Card without the need for a separate typedef statement.

Good Programming Practice 10.4
Capitalize the first letter of typedef names to emphasize that they are synonyms for other type names.
Card can now be used to declare variables of type struct card. The declaration

```c
Card deck[52];
```

declares an array of 52 Card structures (i.e., variables of type struct card). Creating a new name with typedef does not create a new type; typedef simply creates a new type name, which may be used as an alias for an existing type name. A meaningful name helps make the program self-documenting. For example, when we read the previous declaration, we know “deck is an array of 52 Cards.”

Often, typedef is used to create synonyms for the basic data types. For example, a program requiring 4-byte integers may use type int on one system and type long on another. Programs designed for portability often use typedef to create an alias for 4-byte integers, such as Integer. The alias Integer can be changed once in the program to make the program work on both systems.

**Portability Tip 10.2**

Use typedef to help make a program more portable.

### 10.7 Example: High-Performance Card Shuffling and Dealing Simulation

The program in Fig. 10.3 is based on the card shuffling and dealing simulation discussed in Chapter 7. The program represents the deck of cards as an array of structures. The program uses high-performance shuffling and dealing algorithms. The output for the high-performance card shuffling and dealing program is shown in Fig. 10.4.

```c
/* Fig. 10.3: fig10_03.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* prototypes */
void fillDeck( Card * const wDeck, const char * wFace[],
               const char * wSuit[] );
void shuffle( Card * const wDeck );
void deal( const Card * const wDeck );

int main( void )
{


```
Chapter 10  C Structures, Unions, Bit Manipulations and Enumerations

/* initialize array of pointers */

/* initialize array of pointers */
const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};
srand( time( NULL ) ); /* randomize */

fillDeck( deck, face, suit ); /* load the deck with Cards */
shuffle( deck ); /* put Cards in random order */
deal( deck ); /* deal all 52 Cards */
return 0; /* indicates successful termination */

/* place strings into Card structures */
void fillDeck( Card * const wDeck, const char * wFace[],
const char * wSuit[] )
{
    int i; /* counter */
    for ( i = 0; i <= 51; i++ ) {
        wDeck[ i ].face = wFace[ i % 13 ];
        wDeck[ i ].suit = wSuit[ i / 13 ];
    } /* end for */
} /* end function fillDeck */

/* shuffle cards */
void shuffle( Card * const wDeck )
{
    int i; /* counter */
    int j; /* variable to hold random value between 0 - 51 */
    Card temp; /* define temporary structure for swapping Cards */
    for ( i = 0; i <= 51; i++ ) {
        j = rand() % 52;
        temp = wDeck[ i ];
        wDeck[ i ] = wDeck[ j ];
        wDeck[ j ] = temp;
    } /* end for */
} /* end function shuffle */

/* deal cards */
void deal( const Card * const wDeck )
{
    int i; /* counter */

Fig. 10.3  |  High-performance card shuffling and dealing simulation. (Part 2 of 3.)
10.8 Unions

In the program, function `fillDeck` (lines 42–52) initializes the Card array in order with Ace through King of each suit. The Card array is passed (in line 36) to function `shuffle` (lines 55–68), where the high-performance shuffling algorithm is implemented. Function `shuffle` takes an array of 52 Card structures as an argument. The function loops through the 52 cards (array subscripts 0 to 51) using a `for` statement in lines 62–67. For each card, a number between 0 and 51 is picked randomly. Next, the current Card structure and the randomly selected Card structure are swapped in the array (lines 64 through 66). A total of 52 swaps are made in a single pass of the entire array, and the array of Card structures is shuffled! This algorithm cannot suffer from indefinite postponement like the shuffling algorithm presented in Chapter 7. Since the Card structures were swapped in place in the array, the high-performance dealing algorithm implemented in function `deal` (lines 71–80) requires only one pass of the array to deal the shuffled cards.

### Common Programming Error 10.7

Forgetting to include the array subscript when referring to individual structures in an array of structures is a syntax error.

### 10.8 Unions

A union is a derived data type—like a structure—with members that share the same storage space. For different situations in a program, some variables may not be relevant, but other variables are—so a union shares the space instead of wasting storage on variables that are not being used. The members of a union can be of any data type. The number of bytes used to store a union must be at least enough to hold the largest member. In most cases,
unions contain two or more data types. Only one member, and thus one data type, can be referenced at a time. It is your responsibility to ensure that the data in a union is referenced with the proper data type.

**Common Programming Error 10.8**
Referencing data in a union with a variable of the wrong type is a logic error.

**Portability Tip 10.3**
If data is stored in a union as one type and referenced as another type, the results are implementation dependent.

**Union Declarations**
A union is declared with keyword `union` in the same format as a structure. The union definition

```c
union number {
    int x;
    double y;
};
```

indicates that `number` is a union type with members `int x` and `double y`. The union definition is normally placed in a header and included in all source files that use the union type.

**Software Engineering Observation 10.1**
As with a `struct` definition, a `union` definition simply creates a new type. Placing a `union` or `struct` definition outside any function does not create a global variable.

**Operations That Can Be Performed on Unions**
The operations that can be performed on a union are the following: assigning a union to another union of the same type, taking the address (`&`) of a union variable, and accessing union members using the structure member operator and the structure pointer operator. Unions may not be compared using operators `==` and `!=` for the same reasons that structures cannot be compared.

**Initializing Unions in Declarations**
In a declaration, a union may be initialized with a value of the same type as the first union member. For example, with the preceding union, the declaration

```c
union number value = { 10 };  // Correct
```

is a valid initialization of union variable `value` because the union is initialized with an `int`, but the following declaration would truncate the floating-point part of the initializer value and normally would produce a warning from the compiler:

```c
union number value = { 1.43 };  // Incorrect
```

**Common Programming Error 10.9**
Comparing unions is a syntax error.
10.8 Unions

Portability Tip 10.4
The amount of storage required to store a union is implementation dependent but will always be at least as large as the largest member of the union.

Portability Tip 10.5
Some unions may not port easily to other computer systems. Whether a union is portable or not often depends on the storage alignment requirements for the union member data types on a given system.

Performance Tip 10.2
Unions conserve storage.

Demonstrating Unions
The program in Fig. 10.5 uses the variable value (line 13) of type union number to display the value stored in the union as both an int and a double. The program output is implementation dependent. The program output shows that the internal representation of a double value can be quite different from the representation of int.

```c
/* Fig. 10.5: fig10_05.c
   An example of a union */
#include <stdio.h>

/* number union definition */
union number {
    int x;
    double y;
}; /* end union number */

int main( void )
{
    union number value; /* define union variable */

    value.x = 100; /* put an integer into the union */
    printf( "%s
%s
%4s
%d
%s
%4s
%f
%4s
\n", "Put a value in the integer member"," and print both members." ,"int:", value.x," double:", value.y );

    value.y = 100.0; /* put a double into the same union */
    printf( "%s
%s
%4s
%d
%s
%4s
%f
%4s
\n", "Put a value in the floating member", "and print both members.", "int:", value.x, "double:", value.y );

    return 0; /* indicates successful termination */
} /* end main */
```

Fig. 10.5 | Displaying the value of a union in both member data types. (Part 1 of 2.)
Chapter 10  C Structures, Unions, Bit Manipulations and Enumerations

10.9 Bitwise Operators

Computers represent all data internally as sequences of bits. Each bit can assume the value 0 or the value 1. On most systems, a sequence of 8 bits forms a byte—the standard storage unit for a variable of type char. Other data types are stored in larger numbers of bytes. The bitwise operators are used to manipulate the bits of integral operands (char, short, int and long; both signed and unsigned). Unsigned integers are normally used with the bitwise operators.

The bitwise operators are bitwise AND (&), bitwise inclusive OR (|), bitwise exclusive OR (^), left shift (<<), right shift (>>) and complement (~). The bitwise AND, bitwise inclusive OR and bitwise exclusive OR operators compare their two operands bit by bit. The bitwise AND operator sets each bit in the result to 1 if the corresponding bit in both operands is 1. The bitwise inclusive OR operator sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) is 1. The bitwise exclusive OR operator sets each bit in the result to 1 if the corresponding bit in exactly one operand is 1. The left-shift operator shifts the bits of its left operand to the left by the number of bits specified in its right operand. The right-shift operator shifts the bits in its left operand to the right by the number of bits specified in its right operand. The bitwise complement operator sets all 0 bits in its operand to 1 in the result and sets all 1 bits to 0 in the result. Detailed discussions of each bitwise operator appear in the following examples. The bitwise operators are summarized in Fig. 10.6.

---

Portability Tip 10.6

Bitwise data manipulations are machine dependent.

---

Put a value in the integer member and print both members.
int:
  100

double:
  -92559592117433136000000000000000000000000000000000000000000000.000000

Put a value in the floating member and print both members.
int:
  0
double:
  100.000000

Fig. 10.5  Displaying the value of a union in both member data types. (Part 2 of 2.)
10.9 Bitwise Operators

Displaying an Unsigned Integer in Bits

When using the bitwise operators, it is useful to print values in their binary representation to illustrate the precise effects of these operators. The program of Fig. 10.7 prints an unsigned integer in its binary representation in groups of eight bits each. For the examples in this section, we assume that unsigned integers are stored in 4 bytes (32 bits) of memory.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td></td>
<td>bitwise inclusive OR</td>
</tr>
<tr>
<td>^</td>
<td>bitwise exclusive OR</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift</td>
</tr>
<tr>
<td>~</td>
<td>one’s complement</td>
</tr>
</tbody>
</table>

Fig. 10.6 | Bitwise operators.

Displaying an Unsigned Integer in Bits

When using the bitwise operators, it is useful to print values in their binary representation to illustrate the precise effects of these operators. The program of Fig. 10.7 prints an unsigned integer in its binary representation in groups of eight bits each. For the examples in this section, we assume that unsigned integers are stored in 4 bytes (32 bits) of memory.

```c
/* Fig. 10.7: fig10_07.c */
#include <stdio.h>

void displayBits(unsigned value); /* prototype */

int main(void)
{
    unsigned x; /* variable to hold user input */

    printf("Enter an unsigned integer: ");
    scanf("%u", &x);
    displayBits(x);
    return 0; /* indicates successful termination */
} /* end main */

/* display bits of an unsigned integer value */
void displayBits(unsigned value)
{
    unsigned c; /* counter */

    /* define displayMask and left shift 31 bits */
    unsigned displayMask = 1 << 31;
```

Fig. 10.7 | Displaying an unsigned integer in bits. (Part 1 of 2.)
Chapter 10  C Structures, Unions, Bit Manipulations and Enumerations

Function `displayBits` (lines 19–39) uses the bitwise AND operator to combine variable `value` with variable `displayMask` (line 32). Often, the bitwise AND operator is used with an operand called a **mask**—an integer value with specific bits set to 1. Masks are used to hide some bits in a value while selecting other bits. In function `displayBits`, mask variable `displayMask` is assigned the value `1 << 31` (10000000 00000000 00000000 00000000).

The left-shift operator shifts the value 1 from the low order (rightmost) bit to the high order (leftmost) bit in `displayMask` and fills in 0 bits from the right. Line 32

```c
putchar( value & displayMask ? '1' : '0' );
```

determines whether a 1 or a 0 should be printed for the current leftmost bit of variable `value`. When `value` and `displayMask` are combined using &

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Bit 1 &amp; Bit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The results of combining two bits with the bitwise AND operator &.

Fig. 10.7  |  Displaying an unsigned integer in bits. (Part 2 of 2.)

Fig. 10.8  |  Results of combining two bits with the bitwise AND operator &.
Making Function `displayBits` More Scalable and Portable
In line 24 of Fig. 10.7, we hard coded the integer 31 to indicate that the value 1 should be shifted to the leftmost bit in the variable `displayMask`. Similarly, in line 29, we hard coded the integer 32 to indicate that the loop should iterate 32 times—one for each bit in variable value. We assumed that `unsigned` integers are always stored in 32 bits (4 bytes) of memory. Many of today’s popular computers use 32-bit word hardware architectures. C programmers tend to work across many hardware architectures, and sometimes `unsigned` integers will be stored in smaller or larger numbers of bits.

The program in Fig. 10.7 can be made more scalable and more portable by replacing the integer 31 in line 24 with the expression

\[ \text{CHAR_BIT} \times \text{sizeof( unsigned )} - 1 \]

and by replacing the integer 32 in line 29 with the the expression

\[ \text{CHAR_BIT} \times \text{sizeof( unsigned )} \]

The symbolic constant `CHAR_BIT` (defined in `<limits.h>`) represents the number of bits in a byte (normally 8). As you learned in Section 7.7, operator `sizeof` determines the number of bytes used to store an object or type. On a computer that uses 32-bit words, the expression `sizeof( unsigned )` evaluates to 4, so the two preceding expressions evaluate to 31 and 32, respectively. On a computer that uses 16-bit words, the `sizeof` expression evaluates to 2 and the two preceding expressions evaluate to 15 and 16, respectively.

Using the Bitwise AND, Inclusive OR, Exclusive OR and Complement Operators
Figure 10.9 demonstrates the use of the bitwise AND operator, the bitwise inclusive OR operator, the bitwise exclusive OR operator and the bitwise complement operator. The program uses function `displayBits` (lines 53–74) to print the `unsigned` integer values. The output is shown in Fig. 10.10.

```c
/* Fig. 10.9: fig10_09.c */
#include <stdio.h>

void displayBits( unsigned value ); /* prototype */

int main( void )
{
    unsigned number1;
    unsigned number2;
    unsigned mask;
    unsigned setBits;
}
```

Fig. 10.9  Bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 1 of 3.)
/* demonstrate bitwise AND (&) */
number1 = 65535;
mask = 1;
printf("The result of combining the following\n");
displayBits( number1 );
displayBits( mask );
printf("using the bitwise AND operator & is\n");
displayBits( number1 & mask );

/* demonstrate bitwise inclusive OR (|) */
number1 = 15;
setBits = 241;
printf("\nThe result of combining the following\n");
displayBits( number1 );
displayBits( setBits );
printf("using the bitwise inclusive OR operator | is\n");
displayBits( number1 | setBits );

/* demonstrate bitwise exclusive OR (^) */
number1 = 139;
number2 = 199;
printf("\nThe result of combining the following\n");
displayBits( number1 );
displayBits( number2 );
printf("using the bitwise exclusive OR operator ^ is\n");
displayBits( number1 ^ number2 );

/* demonstrate bitwise complement (~)*/
number1 = 21845;
printf("\nThe one's complement of\n");
displayBits( number1 );
printf(" is\n");
displayBits( ~number1 );
return 0; /* indicates successful termination */
} /* end main */

/* display bits of an unsigned integer value */
void displayBits( unsigned value )
{
    unsigned c; /* counter */
/* declare displayMask and left shift 31 bits */
    unsigned displayMask = 1 << 31;
    printf("%10u = ", value);
/* loop through bits */
    for ( c = 1; c <= 32; c++ )
    {
        putchar( value & displayMask ? '1' : '0' );
        value <<= 1; /* shift value left by 1 */
    }
}

Fig. 10.9  |  Bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 2 of 3.)
In Fig. 10.9, integer variable \texttt{number1} is assigned value 65535 (00000000 00000000 11111111 11111111) in line 16 and variable \texttt{mask} is assigned the value 1 (00000000 00000000 00000000 00000001) in line 17. When \texttt{number1} and \texttt{mask} are combined using the bitwise AND operator (&) in the expression \texttt{number1 & mask} (line 22), the result is 1 (00000000 00000000 00000000 00000001). All the bits except the low-order bit in variable \texttt{number1} are “masked off” (hidden) by “ANDing” with variable \texttt{mask}.

The bitwise inclusive OR operator is used to set specific bits to 1 in an operand. In Fig. 10.9, variable \texttt{number1} is assigned 15 (00000000 00000000 00000000 00001111) in line 25, and variable \texttt{setBits} is assigned 241 (00000000 00000000 00000000 11110001) in line 26. When \texttt{number1} and \texttt{setBits} are combined using the bitwise OR operator in the expression \texttt{number1 | setBits} (line 31), the result is 255 (00000000 00000000 00000000 11111111). Figure 10.11 summarizes the results of combining two bits with the bitwise inclusive OR operator.

---

**Common Programming Error 10.11**

*Using the logical OR operator (||) for the bitwise OR operator (|) and vice versa is an error.*
Chapter 10  C Structures, Unions, Bit Manipulations and Enumerations

The bitwise exclusive OR operator (\(^\)) sets each bit in the result to 1 if exactly one of the corresponding bits in its two operands is 1. In Fig. 10.9, variables number1 and number2 are assigned the values 139 (00000000 00000000 00000000 10001011) and 199 (00000000 00000000 00000000 11000111) in lines 34–35. When these variables are combined with the exclusive OR operator in the expression number1 ^ number2 (line 40), the result is 00000000 00000000 00000000 01001100. Figure 10.12 summarizes the results of combining two bits with the bitwise exclusive OR operator.

The bitwise complement operator (~) sets all 1 bits in its operand to 0 in the result and sets all 0 bits to 1 in the result—otherwise referred to as “taking the one’s complement of the value.” In Fig. 10.9, variable number1 is assigned the value 21845 (00000000 00000000 01010101 01010101) in line 43. When the expression ~number1 (line 47) is evaluated, the result is 00000000 00000000 10101010 10101010.

Using the Bitwise Left- and Right-Shift Operators
The program of Fig. 10.13 demonstrates the left-shift operator (<<) and the right-shift operator (>>). Function displayBits is used to print the unsigned integer values.
```c
int main( void )
{
    unsigned number1 = 960; /* initialize number1 */

    /* demonstrate bitwise left shift */
    printf( "\nThe result of left shifting\n" );
    displayBits( number1 );
    printf( "8 bit positions using the " );
    printf( "left shift operator << is\n" );
    displayBits( number1 << 8 );

    /* demonstrate bitwise right shift */
    printf( "\nThe result of right shifting\n" );
    displayBits( number1 );
    printf( "8 bit positions using the " );
    printf( "right shift operator >> is\n" );
    displayBits( number1 >> 8 );
    return 0; /* indicates successful termination */
} /* end main */

/* display bits of an unsigned integer value */
void displayBits( unsigned value )
{
    unsigned c; /* counter */

    /* declare displayMask and left shift 31 bits */
    unsigned displayMask = 1 << 31;

    printf( "%7u = ", value );

    /* loop through bits */
    for ( c = 1; c <= 32; c++ )
    {
        putchar( value & displayMask ? '1' : '0' );
        value <<= 1; /* shift value left by 1 */

        if ( c % 8 == 0 ) { /* output a space after 8 bits */
            putchar( ' ' );
        } /* end if */
    } /* end for */
    putchar( '\n' );
} /* end function displayBits */
```

The result of left shifting
960 = 00000000 00000000 00000011 11000000
8 bit positions using the left shift operator << is
245760 = 00000000 00000011 11000000 00000000

The result of right shifting
960 = 00000000 00000000 00000011 11000000
8 bit positions using the right shift operator >> is
3 = 00000000 00000000 00000000 00000011

Fig. 10.13 | Bitwise shift operators. (Part 2 of 2.)
The left-shift operator (<<) shifts the bits of its left operand to the left by the number of bits specified in its right operand. Bits vacated to the right are replaced with 0s; 1s shifted off the left are lost. In Fig. 10.13, variable number1 is assigned the value 960 (00000000 00000000 00000011 11000000) in line 9. The result of left shifting variable number1 8 bits in the expression number1 << 8 (line 16) is 49152 (00000000 00000000 11000000 00000000).

The right-shift operator (>>) shifts the bits of its left operand to the right by the number of bits specified in its right operand. Performing a right shift on an unsigned integer causes the vacated bits at the left to be replaced by 0s; 1s shifted off the right are lost. In Fig. 10.13, the result of right shifting number1 in the expression number1 >> 8 (line 23) is 3 (00000000 00000000 00000000 00000011).

Common Programming Error 10.12
The result of shifting a value is undefined if the right operand is negative or if the right operand is larger than the number of bits in which the left operand is stored.

Portability Tip 10.7
Right shifting is machine dependent. Right shifting a signed integer fills the vacated bits with 0s on some machines and with 1s on others.

Bitwise Assignment Operators
Each binary bitwise operator has a corresponding assignment operator. These bitwise assignment operators are shown in Fig. 10.14 and are used in a manner similar to the arithmetic assignment operators introduced in Chapter 3.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;=</td>
<td>left to right</td>
<td>highest</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>right to left</td>
</tr>
<tr>
<td>^=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 10.14 | The bitwise assignment operators.

Figure 10.15 shows the precedence and associativity of the various operators introduced to this point in the text. They are shown top to bottom in decreasing order of precedence.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>() [ ] . -&gt;</td>
<td>left to right</td>
<td>highest</td>
</tr>
<tr>
<td>+ - ++ -- ! &amp; * ~ sizeof (type)</td>
<td>right to left</td>
<td>unary</td>
</tr>
</tbody>
</table>
C enables you to specify the number of bits in which an unsigned or int member of a structure or union is stored. This is referred to as a bit field. Bit fields enable better memory utilization by storing data in the minimum number of bits required. Bit field members must be declared as int or unsigned.

Consider the following structure definition:

```c
struct bitCard {
    unsigned face : 4;
    unsigned suit : 2;
    unsigned color : 1;
};
```

which contains three unsigned bit fields—face, suit and color—used to represent a card from a deck of 52 cards. A bit field is declared by following an unsigned or int member name with a colon (:) and an integer constant representing the width of the field (i.e., the number of bits in which the member is stored). The constant representing the width must be an integer between 0 and the total number of bits used to store an int on your system, inclusive. Our examples were tested on a computer with 4-byte (32-bit) integers.

The preceding structure definition indicates that member face is stored in 4 bits, member suit is stored in 2 bits and member color is stored in 1 bit. The number of bits is based on the desired range of values for each structure member. Member face stores values from 0 (Ace) through 12 (King)—4 bits can store values in the range 0–15. Member
suit stores values from 0 through 3 (0 = Diamonds, 1 = Hearts, 2 = Clubs, 3 = Spades)—2 bits can store values in the range 0–3. Finally, member color stores either 0 (Red) or 1 (Black)—1 bit can store either 0 or 1.

Figure 10.16 (output shown in Fig. 10.17) creates array deck containing 52 struct bitCard structures in line 20. Function fillDeck (lines 28–38) inserts the 52 cards in the deck array and function deal (lines 42–54) prints the 52 cards. Notice that bit field members of structures are accessed exactly as any other structure member. Member color is included as a means of indicating the card color on a system that allows color displays. It is possible to specify an unnamed bit field to be used as padding in the structure. For example, the structure definition

```c
struct example {
    unsigned a : 13;
    unsigned : 19;
    unsigned b : 4;
};
```

uses an unnamed 19-bit field as padding—nothing can be stored in those 19 bits. Member b (on our 4-byte-word computer) is stored in another storage unit.

```c
/* Fig. 10.16: fig10_16.c */
/* Representing cards with bit fields in a struct */
#include <stdio.h>
/* bitCard structure definition with bit fields */
struct bitCard {
    unsigned face : 4; /* 4 bits; 0-15 */
    unsigned suit : 2; /* 2 bits; 0-3 */
    unsigned color : 1; /* 1 bit; 0-1 */
}; /* end struct bitCard */
typedef struct bitCard Card; /* new type name for struct bitCard */

void fillDeck( Card * const wDeck ); /* prototype */
void deal( const Card * const wDeck ); /* prototype */

int main( void )
{
    Card deck[ 52 ]; /* create array of Cards */
    fillDeck( deck );
    deal( deck );
    return 0; /* indicates successful termination */
} /* end main */

/* initialize Cards */
void fillDeck( Card * const wDeck )
{
    int i; /* counter */
```
/* loop through wDeck */
for ( i = 0; i <= 51; i++ ) {
    wDeck[ i ].face = i % 13;
    wDeck[ i ].suit = i / 13;
    wDeck[ i ].color = i / 26;
} /* end for */

/* output cards in two column format; cards 0-25 subscripted with
k1 (column 1); cards 26-51 subscripted k2 (column 2) */
void deal( const Card * const wDeck )
{
    int k1; /* subscripts 0-25 */
    int k2; /* subscripts 26-51 */

    /* loop through wDeck */
    for ( k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
        printf( "Card:%3d  Suit:%2d  Color:%2d   ",
                wDeck[ k1 ].face, wDeck[ k1 ].suit, wDeck[ k1 ].color );
        printf( "Card:%3d  Suit:%2d  Color:%2d\n",
                wDeck[ k2 ].face, wDeck[ k2 ].suit, wDeck[ k2 ].color );
    } /* end for */
} /* end function deal */

Fig. 10.16 | Bit fields to store a deck of cards. (Part 2 of 2.)

Card:  0  Suit: 0  Color: 0  Card:  0  Suit: 2  Color: 1
Card:  1  Suit: 0  Color: 0  Card:  1  Suit: 2  Color: 1
Card:  2  Suit: 0  Color: 0  Card:  2  Suit: 2  Color: 1
Card:  3  Suit: 0  Color: 0  Card:  3  Suit: 2  Color: 1
Card:  4  Suit: 0  Color: 0  Card:  4  Suit: 2  Color: 1
Card:  5  Suit: 0  Color: 0  Card:  5  Suit: 2  Color: 1
Card:  6  Suit: 0  Color: 0  Card:  6  Suit: 2  Color: 1
Card:  7  Suit: 0  Color: 0  Card:  7  Suit: 2  Color: 1
Card:  8  Suit: 0  Color: 0  Card:  8  Suit: 2  Color: 1
Card:  9  Suit: 0  Color: 0  Card:  9  Suit: 2  Color: 1
Card: 10  Suit: 0  Color: 0  Card: 10  Suit: 2  Color: 1
Card: 11  Suit: 0  Color: 0  Card: 11  Suit: 2  Color: 1
Card: 12  Suit: 0  Color: 0  Card: 12  Suit: 2  Color: 1
Card:  0  Suit: 1  Color: 0  Card:  0  Suit: 3  Color: 1
Card:  1  Suit: 1  Color: 0  Card:  1  Suit: 3  Color: 1
Card:  2  Suit: 1  Color: 0  Card:  2  Suit: 3  Color: 1
Card:  3  Suit: 1  Color: 0  Card:  3  Suit: 3  Color: 1
Card:  4  Suit: 1  Color: 0  Card:  4  Suit: 3  Color: 1
Card:  5  Suit: 1  Color: 0  Card:  5  Suit: 3  Color: 1
Card:  6  Suit: 1  Color: 0  Card:  6  Suit: 3  Color: 1
Card:  7  Suit: 1  Color: 0  Card:  7  Suit: 3  Color: 1
Card:  8  Suit: 1  Color: 0  Card:  8  Suit: 3  Color: 1
Card:  9  Suit: 1  Color: 0  Card:  9  Suit: 3  Color: 1
Card: 10  Suit: 1  Color: 0  Card: 10  Suit: 3  Color: 1
Card: 11  Suit: 1  Color: 0  Card: 11  Suit: 3  Color: 1
Card: 12  Suit: 1  Color: 0  Card: 12  Suit: 3  Color: 1

Fig. 10.17 | Output of the program in Fig. 10.16.
An unnamed bit field with a zero width is used to align the next bit field on a new storage-unit boundary. For example, the structure definition

```c
struct example {
    unsigned a : 13;
    unsigned : 0;
    unsigned b : 4;
};
```

uses an unnamed 0-bit field to skip the remaining bits (as many as there are) of the storage unit in which `a` is stored and to align `b` on the next storage-unit boundary.

### Portability Tip 10.8
Bit-field manipulations are machine dependent. For example, some computers allow bit fields to cross word boundaries, whereas others do not.

### Common Programming Error 10.13
Attempting to access individual bits of a bit field as if they were elements of an array is a syntax error. Bit fields are not “arrays of bits.”

### Common Programming Error 10.14
Attempting to take the address of a bit field (the & operator may not be used with bit fields because they do not have addresses).

### Performance Tip 10.4
Although bit fields save space, using them can cause the compiler to generate slower-executing machine-language code. This occurs because it takes extra machine language operations to access only portions of an addressable storage unit. This is one of many examples of the kinds of space–time trade-offs that occur in computer science.

## 10.11 Enumeration Constants

C provides one final user-defined type called an enumeration. An enumeration, introduced by the keyword `enum`, is a set of integer enumeration constants represented by identifiers. Values in an `enum` start with 0, unless specified otherwise, and are incremented by 1. For example, the enumeration

```c
enum months {
    JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
};
```

creates a new type, `enum months`, in which the identifiers are set to the integers 0 to 11, respectively. To number the months 1 to 12, use the following enumeration:

```c
enum months {
    JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
};
```

Since the first value in the preceding enumeration is explicitly set to 1, the remaining values are incremented from 1, resulting in the values 1 through 12. The identifiers in an enumeration must be unique. The value of each enumeration constant of an enumeration can be set explicitly in the definition by assigning a value to the identifier. Multiple members of an enumeration can have the same constant value. In the program of Fig. 10.18,
the enumeration variable month is used in a for statement to print the months of the year from the array monthName. We’ve made monthName[0] the empty string "". Some programmers might prefer to set monthName[0] to a value such as ***ERROR*** to indicate that a logic error occurred.

**Common Programming Error 10.15**

Assigning a value to an enumeration constant after it has been defined is a syntax error.

**Good Programming Practice 10.5**

Use only uppercase letters enumeration constant names. This makes these constants stand out in a program and reminds you that enumeration constants are not variables.

```c
/* Fig. 10.18: fig10_18.c  
Using an enumeration type */
#include <stdio.h>

/* enumeration constants represent months of the year */
enum months {
    JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
};

int main( void )
{
    enum months month; /* can contain any of the 12 months */

    /* initialize array of pointers */
    const char *monthName[] = {
        "", "January", "February", "March", 
        "November", "December" }
;
    /* loop through months */
    for ( month = JAN; month <= DEC; month++ ) {
        printf("%2d%11s\n", month, monthName[ month ] );
    } /* end for */
    return 0; /* indicates successful termination */
} /* end main */
```

**Fig. 10.18**  |  Using an enumeration.
Summary

Section 10.1 Introduction
- Structures are collections of related variables under one name. They may contain variables of many different data types.
- Structures are commonly used to define records to be stored in files.
- Pointers and structures facilitate the formation of more complex data structures such as linked lists, queues, stacks and trees.

Section 10.2 Structure Definitions
- **Keyword** `struct` introduces a structure definition.
- The identifier following keyword `struct` is the structure tag, which names the structure definition. The structure tag is used with the keyword `struct` to declare variables of the structure type.
- Variables declared within the braces of the structure definition are the structure's members.
- Members of the same structure type must have unique names.
- Each structure definition must end with a semicolon.
- Structure members can be variables of the primitive data types or aggregates, such as arrays and other structures.
- A structure cannot contain an instance of itself but may include a pointer to another object of the same type.
- A structure containing a member that is a pointer to the same structure type is referred to as a self-referential structure. Self-referential structures are used to build linked data structures.
- Structure definitions do not reserve any space in memory; they create new data types that are used to define variables.
- Variables of a given structure type can be declared by placing a comma-separated list of variable names between the closing brace of the structure definition and its ending semicolon.
- The structure tag name is optional. If a structure definition does not contain a structure tag name, variables of the structure type may be declared only in the structure definition.
- The only valid operations that may be performed on structures are assigning structure variables to variables of the same type, taking the address (`&`) of a structure variable, accessing the members of a structure variable and using the `sizeof` operator to determine the size of a structure variable.

Section 10.3 Initializing Structures
- Structures can be initialized using initializer lists.
- If there are fewer initializers in the list than members in the structure, the remaining members are automatically initialized to 0 (or `NULL` if the member is a pointer).
- Members of structure variables defined outside a function definition are initialized to 0 or `NULL` if they are not explicitly initialized in the external definition.
- Structure variables may be initialized in assignment statements by assigning a structure variable of the same type, or by assigning values to the individual members of the structure.

Section 10.4 Accessing Structure Members
- The structure member operator (`.`) and the structure pointer operator (`->`) are used to access structure members.
- The structure member operator accesses a structure member via the structure variable name.
- The structure pointer operator accesses a structure member via a pointer to the structure.
Section 10.5 Using Structures with Functions

• Structures may be passed to functions by passing individual structure members, by passing an entire structure or by passing a pointer to a structure.
• Structure variables are passed by value by default.
• To pass a structure by reference, pass its address. Arrays of structures—like all other arrays—are automatically passed by reference.
• To pass an array by value, create a structure with the array as a member. Structures are passed by value, so the array is passed by value.

Section 10.6 typedef

• The keyword typedef provides a mechanism for creating synonyms for previously defined types.
• Names for structure types are often defined with typedef to create shorter type names.
• Often, typedef is used to create synonyms for the basic data types. For example, a program requiring 4-byte integers may use type int on one system and type long on another. Programs designed for portability often use typedef to create an alias for 4-byte integers such as Integer. The alias Integer can be changed once in the program to make the program work on both systems.

Section 10.8 Unions

• A union is declared with keyword union in the same format as a structure. Its members share the same storage space.
• The members of a union can be of any data type. The number of bytes used to store a union must be at least enough to hold the largest member.
• Only one member of a union can be referenced at a time. It is your responsibility to ensure that the data in a union is referenced with the proper data type.
• The operations that can be performed on a union are assigning a union to another of the same type, taking the address (&) of a union variable, and accessing union members using the structure member operator and the structure pointer operator.
• A union may be initialized in a declaration with a value of the same type as the first union member.

Section 10.9 Bitwise Operators

• Computers represent all data internally as sequences of bits with the values 0 or 1.
• On most systems, a sequence of 8 bits form a byte—the standard storage unit for a variable of type char. Other data types are stored in larger numbers of bytes.
• The bitwise operators are used to manipulate the bits of integral operands (char, short, int and long; both signed and unsigned). Unsigned integers are normally used.
• The bitwise operators are bitwise AND (&), bitwise inclusive OR (|), bitwise exclusive OR (^), left shift (<<), right shift (>>) and complement (~).
• The bitwise AND, bitwise inclusive OR and bitwise exclusive OR operators compare their two operands bit by bit. The bitwise AND operator sets each bit in the result to 1 if the corresponding bit in both operands is 1. The bitwise inclusive OR operator sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) is 1. The bitwise exclusive OR operator sets each bit in the result to 1 if the corresponding bit in exactly one operand is 1.
• The left-shift operator shifts the bits of its left operand to the left by the number of bits specified in its right operand. Bits vacated to the right are replaced with 0s; 1s shifted off the left are lost.
• The right-shift operator shifts the bits in its left operand to the right by the number of bits specified in its right operand. Performing a right shift on an unsigned integer causes the vacated bits at the left to be replaced by 0s; bits shifted off the right are lost.
• The bitwise complement operator sets all 0 bits in its operand to 1 in the result and sets all 1 bits to 0 in the result.

• Often, the bitwise AND operator is used with an operand called a mask—an integer value with specific bits set to 1. Masks are used to hide some bits in a value while selecting other bits.

• The symbolic constant CHAR_BIT (defined in `<limits.h>`) represents the number of bits in a byte (normally 8). It can be used to make a bit-manipulation program more scalable and portable.

• Each binary bitwise operator has a corresponding assignment operator.

Section 10.10 Bit Fields
• C enables you to specify the number of bits in which an unsigned or int member of a structure or union is stored. This is referred to as a bit field. Bit fields enable better memory utilization by storing data in the minimum number of bits required.

• A bit field is declared by following an unsigned or int member name with a colon (:) and an integer constant representing the width of the field. The constant must be an integer between 0 and the total number of bits used to store an int on your system, inclusive.

• Bit-field members of structures are accessed exactly as any other structure member.

• It is possible to specify an unnamed bit field to be used as padding in the structure.

• An unnamed bit field with a zero width aligns the next bit field on a new storage unit boundary.

Section 10.11 Enumeration Constants
• An enum defines a set of integer constants represented by identifiers. Values in an enum start with 0, unless specified otherwise, and are incremented by 1.

• The identifiers in an enum must be unique.

• The value of an enum constant can be set explicitly via assignment in the enum definition.

• Multiple members of an enumeration can have the same constant value.

Terminology

structure member operator 386
bitwise complement operator 400
aggregate 383
arrow operator (->) 386
bit field 403
bit field member name 403
bitwise AND (&) operator 394
bitwise assignment operator 402
bitwise complement operator (~) 400
bitwise exclusive OR (^) operator 394
bitwise inclusive OR (|) operator 394
CHAR_BIT symbolic constant 397
complement operator (~) 394
derived data type 383
enumeration 406
enumeration constants 406
left-shift operator (<<) 394
mask 396
member 383
member name (bit field) 403
one's complement 400
padding 404
pointer to the structure 386
right-shift operator (>>) 394
self-referential structure 384
struct 383
structure 383
structure member (.) operator 386
structure pointer (->) operator 386
structure tag 383
structure type 383
typedef 388
union 391
unnamed bit field 404
unnamed bit field with a zero width 406
width of a bit field 403
Self-Review Exercises

10.1 Fill in the blanks in each of the following:

a) A(n) ________ is a collection of related variables under one name.

b) A(n) ________ is a collection of variables under one name in which the variables share

the same storage.

c) The bits in the result of an expression using the ________ operator are set to 1 if the

 corresponding bits in each operand are set to 1. Otherwise, the bits are set to zero.

d) The variables declared in a structure definition are called its ________.

e) In an expression using the ________ operator, bits are set to 1 if at least one of the cor-

 responding bits in either operand is set to 1. Otherwise, the bits are set to zero.

f) Keyword ________ introduces a structure declaration.

g) Keyword ________ is used to create a synonym for a previously defined data type.

h) In an expression using the ________ operator, bits are set to 1 if exactly one of the cor-

 responding bits in either operand is set to 1. Otherwise, the bits are set to zero.

i) The bitwise AND operator (&) is often used to ________ bits, that is to select certain bits

 while zeroing others.

j) Keyword ________ is used to introduce a union definition.

k) The name of the structure is referred to as the structure ________.

l) A structure member is accessed with either the ________ or the ________ operator.

m) The ________ and ________ operators are used to shift the bits of a value to the left or

 to the right, respectively.

n) A(n) ________ is a set of integers represented by identifiers.

10.2 State whether each of the following is true or false. If false, explain why.

a) Structures may contain variables of only one data type.

b) Two unions can be compared (using ==) to determine if they are equal.

c) The tag name of a structure is optional.

d) Members of different structures must have unique names.

e) Keyword typedef is used to define new data types.

f) Structures are always passed to functions by reference.

g) Structures may not be compared by using operators == and !=.

10.3 Write code to accomplish each of the following:

a) Define a structure called part containing int variable partNumber and char array part-

 Name with values that may be as long as 25 characters (including the terminating null

 character).

b) Define Part to be a synonym for the type struct part.

c) Use Part to declare variable a to be of type struct part, array b[ 10 ] to be of type

 struct part and variable ptr to be of type pointer to struct part.

d) Read a part number and a part name from the keyboard into the individual members

 of variable a.

e) Assign the member values of variable a to element 3 of array b.

f) Assign the address of array b to the pointer variable ptr.

g) Print the member values of element 3 of array b using the variable ptr and the structure

 pointer operator to refer to the members.

10.4 Find the error in each of the following:

a) Assume that struct card has been defined containing two pointers to type char, namely

 face and suit. Also, the variable c has been defined to be of type struct card and the

 variable cPtr has been defined to be of type pointer to struct card. Variable cPtr

 has been assigned the address of c.

 printf( "%s\n", *cPtr->face );
b) Assume that `struct card` has been defined containing two pointers to type `char`, namely `face` and `suit`. Also, the array `hearts[13]` has been defined to be of type `struct card`. The following statement should print the member `face` of array element 10.

```c
printf("%s\n", hearts.face);
```

c) ```c
union values {
    char w;
    float x;
    double y;
};
``` 

d) ```c
union values v = {1.27};
``` 

e) Assume `struct person` has been defined as in part (d) but with the appropriate correction.

g) ```c
struct person {
    char lastName[15];
    char firstName[15];
    int age;
};
``` 

e) Assume `struct person` has been defined as in part (d) but with the appropriate correction.

g) ```
person d;
``` 
f) Assume variable `p` has been declared as type `struct person` and variable `c` has been declared as type `struct card`.

```c
p = c;
``` 

**Answers to Self-Review Exercises**

**10.1**

a) structure.  
b) union.  
c) bitwise AND (`&`).  
d) members.  
e) bitwise inclusive OR (`|`).  
f) `struct`.  
g) `typedef`.  
h) bitwise exclusive OR (`^`).  
i) mask.  
j) `union`.  
k) tag name.  
l) `structure member`, `structure pointer`.  
m) left-shift operator (`<<`), right-shift operator (`>>`).  
n) `enumeration`.

**10.2**

a) False. A structure can contain variables of many data types.

b) False. Unions cannot be compared because there might be bytes of undefined data with different values in union variables that are otherwise identical.

c) True.

d) False. The members of separate structures can have the same names, but the members of the same structure must have unique names.

e) False. Keyword `typedef` is used to define new names (synonyms) for previously defined data types.

f) False. Structures are always passed to functions call-by-value.

g) True, because of alignment problems.

**10.3**

a) ```c
struct part {
    int partNumber;
    char partName[26];
};
``` 
b) ```c
typedef struct part Part;
``` 
c) ```c
Part a, b[10], *ptr;
``` 
d) ```c
scanf("%d%25s", &a.partNumber, &a.partName);
``` 
e) ```c
b[3] = a;
``` 
f) ```c
ptr = b;
``` 
g) ```c
printf("%d %s\n", (ptr + 3)->partNumber, (ptr + 3)->partName);
```
Exercises

10.4
a) The parentheses that should enclose \*cPtr have been omitted, causing the order of evaluation of the expression to be incorrect. The expression should be 
\((\ast cPtr)\rightarrow face\)
b) The array subscript has been omitted. The expression should be 
hearts[10].face.
c) A union can be initialized only with a value that has the same type as the union’s first member.
d) A semicolon is required to end a structure definition.
e) Keyword struct was omitted from the variable declaration. The declaration should be 
struct person d;
f) Variables of different structure types cannot be assigned to one another.

Exercises

10.5
Provide the definition for each of the following structures and unions:
a) Structure inventory containing character array partName[30], integer partNumber, floating point price, integer stock and integer reorder.
b) Union data containing char c, short s, long b, float f and double d.
c) A structure called address that contains character arrays
streetAddress[25], city[20], state[3] and zipCode[6].
d) Structure student that contains arrays firstName[15] and lastName[15] and variable homeAddress of type struct address from part (c).
e) Structure test containing 16 bit fields with widths of 1 bit. The names of the bit fields are the letters a to p.

10.6
Given the following structure and variable definitions,

```c
struct customer {
    char lastName[15];
    char firstName[15];
    int customerNumber;

    struct {
        char phoneNumber[11];
        char address[50];
        char city[15];
        char state[3];
        char zipCode[6];
    } personal;
}
customerRecord, *customerPtr;
```

customerPtr = &customerRecord;

write an expression that can be used to access the structure members in each of the following parts:
a) Member lastName of structure customerRecord.
b) Member lastName of the structure pointed to by customerPtr.
c) Member firstName of structure customerRecord.
d) Member firstName of the structure pointed to by customerPtr.
e) Member customerNumber of structure customerRecord.
f) Member customerNumber of the structure pointed to by customerPtr.
g) Member phoneNumber of member personal of structure customerRecord.
h) Member phoneNumber of member personal of the structure pointed to by customerPtr.
i) Member address of member personal of structure customerRecord.
j) Member address of member personal of the structure pointed to by customerPtr.
k) Member city of member personal of structure customerRecord.
l) Member city of member personal of the structure pointed to by customerPtr.
Chapter 10  C Structures, Unions, Bit Manipulations and Enumerations

m) Member state of member personal of structure customerRecord.

n) Member state of member personal of the structure pointed to by customerPtr.

o) Member zipCode of member personal of structure customerRecord.

p) Member zipCode of member personal of the structure pointed to by customerPtr.

10.7  (Card Shuffling and Dealing Modification) Modify the program of Fig. 10.16 to shuffle the cards using a high-performance shuffle (as shown in Fig. 10.3). Print the resulting deck in two-column format as in Fig. 10.4. Precede each card with its color.

10.8  (Using Unions) Create union integer with members char c, short s, int i and long l. Write a program that inputs value of type char, short, int and long and stores the values in union variables of type union integer. Each union variable should be printed as a char, a short, an int and a long. Do the values always print correctly?

10.9  (Using Unions) Create union floatingPoint with members float f, double d and long double x. Write a program that inputs value of type float, double and long double and stores the values in union variables of type union floatingPoint. Each union variable should be printed as a float, a double and a long double. Do the values always print correctly?

10.10 (Right-Shifting Integers) Write a program that right shifts an integer variable 4 bits. The program should print the integer in bits before and after the shift operation. Does your system place 0s or 1s in the vacated bits?

10.11 (Right-Shifting Integers) If your computer uses 2-byte integers, modify the program of Fig. 10.7 so that it works with 2-byte integers.

10.12 (Left-Shifting Integers) Left shifting an unsigned integer by 1 bit is equivalent to multiplying the value by 2. Write function power2 that takes two integer arguments number and pow and calculates

\[ \text{number} \times 2^{\text{pow}} \]

Use the shift operator to calculate the result. Print the values as integers and as bits.

10.13 (Packing Characters into an Integer) The left-shift operator can be used to pack two character values into an unsigned integer variable. Write a program that inputs two characters from the keyboard and passes them to function packCharacters. To pack two characters into an unsigned integer variable, assign the first character to the unsigned variable, shift the unsigned variable left by 8 bit positions and combine the unsigned variable with the second character using the bitwise inclusive OR operator. The program should output the characters in their bit format before and after they are packed into the unsigned integer to prove that the characters are in fact packed correctly in the unsigned variable.

10.14 (Unpacking Characters from an Integer) Using the right-shift operator, the bitwise AND operator and a mask, write function unpackCharacters that takes the unsigned integer from Exercise 10.13 and unpacks it into two characters. To unpack two characters from an unsigned integer, combine the unsigned integer with the mask \( 65280 \) (\( 00000000 \ 00000000 \ 11111111 \ 00000000 \)) and right shift the result 8 bits. Assign the resulting value to a char variable. Then combine the unsigned integer with the mask \( 255 \) (\( 00000000 \ 00000000 \ 00000000 \ 11111111 \)) and assign the result to another char variable. The program should print the unsigned integer in bits before it is unpacked, then print the characters in bits to confirm that they were unpacked correctly.

10.15 (Packing Characters into an Integer) If your system uses 4-byte integers, rewrite the program of Exercise 10.13 to pack 4 characters.

10.16 (Unpacking Characters from an Integer) If your system uses 4-byte integers, rewrite the function unpackCharacters of Exercise 10.14 to unpack 4 characters. Create the masks you need to
unpack the 4 characters by left shifting the value 255 in the mask variable by 8 bits 0, 1, 2 or 3 times (depending on the byte you’re unpacking).

10.17 (Reversing the Order of an Integer’s Bits) Write a program that reverses the order of the bits in an unsigned integer value. The program should input the value from the user and call function reverseBits to print the bits in reverse order. Print the value in bits both before and after the bits are reversed to confirm that the bits are reversed properly.

10.18 (Portable displayBits Function) Modify function displayBits of Fig. 10.7 so it is portable between systems using 2-byte integers and systems using 4-byte integers. [Hint: Use the sizeof operator to determine the size of an integer on a particular machine.]

10.19 (What is the Value of X?) The following program uses function multiple to determine if the integer entered from the keyboard is a multiple of some integer X. Examine the function multiple, then determine X’s value.

```c
/* ex10_19.c */
/* This program determines if a value is a multiple of X. */
#include <stdio.h>

int multiple( int num ); /* prototype */

int main( void )
{
  int y; /* y will hold an integer entered by the user */

  printf( "Enter an integer between 1 and 32000: " );
  scanf( "%d", &y );

  /* if y is a multiple of X */
  if ( multiple( y ) ) {
    printf( "%d is a multiple of X\n", y );
  } /* end if */
  else {
    printf( "%d is not a multiple of X\n", y );
  } /* end else */

  return 0; /* indicates successful termination */
} /* end main */

/* determine if num is a multiple of X */
int multiple( int num )
{
  int i; /* counter */
  int mask = 1; /* initialize mask */
  int mult = 1; /* initialize mult */

  for ( i = 1; i <= 10; i++, mask <<= 1 ) {
    if ( ( num & mask ) != 0 ) {
      mult = 0;
      break;
    } /* end if */
  } /* end for */

  return mult;
} /* end function multiple */
```
Chapter 10  C Structures, Unions, Bit Manipulations and Enumerations

10.20 What does the following program do?

```c
/* ex10_20.c */
#include <stdio.h>

int mystery(unsigned bits); /* prototype */

int main(void)
{
    unsigned x; /* x will hold an integer entered by the user */
    printf("Enter an integer: ");
    scanf("%u", &x);
    printf("The result is %d\n", mystery(x));
    return 0; /* indicates successful termination */
} /* end main */

/* What does this function do? */
int mystery(unsigned bits)
{
    unsigned i; /* counter */
    unsigned mask = 1 << 31; /* initialize mask */
    unsigned total = 0; /* initialize total */
    for (i = 1; i <= 32; i++, bits <<= 1) {
        if ((bits & mask) == mask) {
            total++;
        } /* end if */
    } /* end for */
    return !(total % 2) ? 1 : 0;
} /* end function mystery */
```

Making a Difference

10.21 (Computerization of Health Records) A health care issue that has been in the news lately is the computerization of health records. This possibility is being approached cautiously because of sensitive privacy and security concerns, among others. Computerizing health records could make it easier for patients to share their health profiles and histories among their various health care professionals. This could improve the quality of health care, help avoid drug conflicts and erroneous drug prescriptions, reduce costs and in emergencies could save lives. In this exercise, you’ll design a “starter” HealthProfile structure for a person. The structure’s members should include the person’s first name, last name, gender, date of birth (consisting of separate attributes for the month, day and year of birth), height (in inches) and weight (in pounds). Your program should have a function that receives this data and uses it to set the members of a HealthProfile variable. The program also should include functions that calculate and return the user’s age in years, maximum heart rate and target-heart-rate range (see Exercise 3.48), and body mass index (BMI; see Exercise 2.32). The program should prompt for the person’s information, create a HealthProfile variable for that person and display the information from that variable—including the person’s first name, last name, gender, date of birth, height and weight—then it should calculate and display the person’s age in years, BMI, maximum heart rate and target-heart-rate range. It should also display the “BMI values” chart from Exercise 2.32.
C File Processing

I read part of it all the way through.
—Samuel Goldwyn

Hats off!
The flag is passing by.
—Henry Holcomb Bennett

Consciousness … does not appear to itself chopped up in bits. … A “river” or a “stream” are the metaphors by which it is most naturally described.
—William James

I can only assume that a “Do Not File” document is filed in a “Do Not File” file.
—Senator Frank Church

Objectives

In this chapter, you’ll learn:

■ To create, read, write and update files.
■ Sequential access file processing.
■ Random-access file processing.
11.1 Introduction

Storage of data in variables and arrays is temporary—such data is lost when a program terminates. Files are used for permanent retention of data. Computers store files on secondary storage devices, especially disk storage devices. In this chapter, we explain how data files are created, updated and processed by C programs. We consider sequential-access files and random-access files.

11.2 Data Hierarchy

Ultimately, all data items processed by a computer are reduced to combinations of zeros and ones. This occurs because it’s simple and economical to build electronic devices that can assume two stable states—one of the states represents 0 and the other represents 1. It’s remarkable that the impressive functions performed by computers involve only the most fundamental manipulations of 0s and 1s.

The smallest data item in a computer can assume the value 0 or the value 1. Such a data item is called a bit (short for “binary digit”—a digit that can assume one of two values). Computer circuitry performs various simple bit manipulations such as determining a bit’s value, setting a bit’s value and reversing a bit (from 1 to 0 or from 0 to 1).

It’s cumbersome to work with data in the low-level form of bits. Instead, programmers prefer to work with data in the form of decimal digits (i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9), letters (i.e., A–Z, and a–z), and special symbols (i.e., $, @, %, &, *, (, ), -, +, ”, :, ?, /, and others). Digits, letters, and special symbols are referred to as characters. The set of all characters that may be used to write programs and represent data items on a particular computer is called that computer’s character set. Since computers can process only 1s and 0s, every character in a computer’s character set is represented as a pattern of 1s and 0s (called a byte). Today, bytes are most commonly composed of eight bits. You create programs and data items as characters; computers then manipulate and process these characters as patterns of bits.

Just as characters are composed of bits, fields are composed of characters. A field is a group of characters that conveys meaning. For example, a field consisting solely of uppercase and lowercase letters can be used to represent a person’s name.

Data items processed by computers form a data hierarchy in which data items become larger and more complex in structure as we progress from bits, to characters (bytes), to fields, and so on.
A record (i.e., a struct in C) is composed of several fields. In a payroll system, for example, a record for a particular employee might consist of the following fields:

1. Social Security number (alphanumeric field)
2. Name (alphabetic field)
3. Address (alphanumeric field)
4. Hourly salary rate (numeric field)
5. Number of exemptions claimed (numeric field)
6. Year-to-date earnings (numeric field)
7. Amount of federal taxes withheld (numeric field)

Thus, a record is a group of related fields. In the preceding example, each of the fields belongs to the same employee. Of course, a particular company may have many employees and will have a payroll record for each employee. A file is a group of related records. A company’s payroll file normally contains one record for each employee. Thus, a payroll file for a small company might contain only 22 records, whereas a payroll file for a large company might contain 100,000 records. It’s not unusual for an organization to have hundreds or even thousands of files, with some containing billions or even trillions of characters of information. Figure 11.1 illustrates the data hierarchy.

To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a record key. A record key identifies a record as belonging to a particular person or entity. For example, in the payroll record described in this section, the Social Security number would normally be chosen as the record key.
There are many ways of organizing records in a file. The most popular type of organization is called a **sequential file**, in which records are typically stored in order by the record key field. In a payroll file, records are usually placed in order by Social Security Number. The first employee record in the file contains the lowest Social Security number, and subsequent records contain increasingly higher Social Security numbers.

Most businesses store data in many different files. For example, companies may have payroll files, accounts receivable files (listing money due from clients), accounts payable files (listing money due to suppliers), inventory files (listing facts about all the items handled by the business) and many other types of files. A group of related files is sometimes called a **database**. A collection of programs designed to create and manage databases is called a **database management system** (DBMS).

### 11.3 Files and Streams

C views each file simply as a sequential stream of bytes (Fig. 11.2). Each file ends either with an **end-of-file marker** or at a specific byte number recorded in a system-maintained, administrative data structure. When a file is opened, a **stream** is associated with the file. Three files and their associated streams are automatically opened when program execution begins—the **standard input**, the **standard output** and the **standard error**. Streams provide communication channels between files and programs. For example, the standard input stream enables a program to read data from the keyboard, and the standard output stream enables a program to print data on the screen. Opening a file returns a pointer to a FILE structure (defined in `<stdio.h>`) that contains information used to process the file. This structure includes a **file descriptor**, i.e., an index into an operating system array called the open file table. Each array element contains a **file control block (FCB)** that the operating system uses to administer a particular file. The standard input, standard output and standard error are manipulated using file pointers stdin, stdout and stderr.

![Fig. 11.2](http://example.com/fig112.png)  
C’s view of a file of n bytes.

The standard library provides many functions for reading data from files and for writing data to files. Function fgetc, like getchar, reads one character from a file. Function fgetc receives as an argument a FILE pointer for the file from which a character will be read. The call fgetc( stdin ) reads one character from stdin—the standard input. This call is equivalent to the call getchar(). Function fputc, like putchar, writes one character to a file. Function fputc receives as arguments a character to be written and a pointer for the file to which the character will be written. The function call fputc('a', stdout ) writes the character 'a' to stdout—the standard output. This call is equivalent to putchar( 'a' ).

Several other functions used to read data from standard input and write data to standard output have similarly named file processing functions. The fgets and fputs functions, for example, can be used to read a line from a file and write a line to a file, respectively. In the next several sections, we introduce the file processing equivalents of
functions \texttt{scanf} and \texttt{printf}—\texttt{fscanf} and \texttt{fprintf}. Later in the chapter we discuss functions \texttt{fread} and \texttt{fwrite}.

11.4 Creating a Sequential-Access File

C imposes no structure on a file. Thus, notions such as a record of a file do not exist as part of the C language. Therefore, you must provide a file structure to meet the requirements of a particular application. The following example shows how to impose a record structure on a file.

Figure 11.3 creates a simple sequential-access file that might be used in an accounts receivable system to help keep track of the amounts owed by a company’s credit clients. For each client, the program obtains an account number, the client’s name and the client’s balance (i.e., the amount the client owes the company for goods and services received in the past). The data obtained for each client constitutes a “record” for that client. The account number is used as the record key in this application—the file will be created and maintained in account number order. This program assumes the user enters the records in account number order. In a comprehensive accounts receivable system, a sorting capability would be provided so the user could enter the records in any order. The records would then be sorted and written to the file. [Note: Figures 11.7–11.8 use the data file created in Fig. 11.3, so you must run Fig. 11.3 before Figs. 11.7–11.8.]

```c
/* Fig. 11.3: fig11_03.c
Create a sequential file */
#include <stdio.h>

int main( void )
{
    int account; /* account number */
    char name[ 30 ]; /* account name */
    double balance; /* account balance */

    FILE *cfPtr; /* cfPtr = clients.dat file pointer */

    /* fopen opens file. Exit program if unable to create file */
    if ( ( cfPtr = fopen( "clients.dat", "w" ) ) == NULL ) {
        printf( "File could not be opened\n" );
    } /* end if */
    else {
        printf( "Enter the account, name, and balance.\n" );
        printf( "Enter EOF to end input.\n" );
        printf( "? " );
        scanf( "%d%s%f", &account, name, &balance );

        /* write account, name and balance into file with fprintf */
        while ( !feof( stdin ) ) {
            fprintf( cfPtr, "%d %s %.2f\n", account, name, balance );
            printf( "? " );
            scanf( "%d%s%f", &account, name, &balance );
        } /* end while */
    }
}
```

Fig. 11.3 | Creating a sequential file. (Part 1 of 2.)
Now let’s examine this program. Line 11 states that cfptr is a pointer to a FILE structure. A C program administers each file with a separate FILE structure. You need not know the specifics of the FILE structure to use files, though the interested reader can study the declaration in stdio.h. We’ll soon see precisely how the FILE structure leads indirectly to the operating system’s file control block (FCB) for a file.

Each open file must have a separately declared pointer of type FILE that is used to refer to the file. Line 14 names the file—"clients.dat"—to be used by the program and establishes a “line of communication” with the file. The file pointer cfPtr is assigned a pointer to the FILE structure for the file opened with fopen. Function fopen takes two arguments: a file name and a file open mode. The file open mode "w" indicates that the file is to be opened for writing. If a file does not exist and it’s opened for writing, fopen creates the file. If an existing file is opened for writing, the contents of the file are discarded without warning. In the program, the if statement is used to determine whether the file pointer cfPtr is NULL (i.e., the file is not opened). If it’s NULL, the program prints an error message and terminates. Otherwise, the program processes the input and writes it to the file.

The program prompts the user to enter the various fields for each record or to enter end-of-file when data entry is complete. Figure 11.4 lists the key combinations for entering end-of-file for various computer systems.

Line 24 uses function feof to determine whether the end-of-file indicator is set for the file to which stdin refers. The end-of-file indicator informs the program that there is no more data to be processed. In Fig. 11.3, the end-of-file indicator is set for the standard input when the user enters the end-of-file key combination. The argument to function fclose( cfPtr ); /* fclose closes file */
return 0; /* indicates successful termination */

Enter the account, name, and balance. Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z

Common Programming Error 11.1
Opening an existing file for writing ("w") when, in fact, the user wants to preserve the file, discards the contents of the file without warning.

Common Programming Error 11.2
Forgetting to open a file before attempting to reference it in a program is a logic error.

The program processes the input and writes it to the file.

Common Programming Error 11.1
Opening an existing file for writing ("w") when, in fact, the user wants to preserve the file, discards the contents of the file without warning.

Common Programming Error 11.2
Forgetting to open a file before attempting to reference it in a program is a logic error.
11.4  Creating a Sequential-Access File

feof is a pointer to the file being tested for the end-of-file indicator (stdin in this case). The function returns a nonzero (true) value when the end-of-file indicator has been set; otherwise, the function returns zero. The while statement that includes the feof call in this program continues executing while the end-of-file indicator is not set.

Line 25 writes data to the file clients.dat. The data may be retrieved later by a program designed to read the file (see Section 11.5). Function fprintf is equivalent to printf except that fprintf also receives as an argument a file pointer for the file to which the data will be written. Function fprintf can output data to the standard output by using stdout as the file pointer, as in:

```c
fprintf( stdout, "%d %s %.2f\n", account, name, balance );
```

**Fig. 11.4**  |  End-of-file key combinations for various popular operating systems.

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Key combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux/Mac OS X/UNIX</td>
<td>&lt;Ctrl&gt; d</td>
</tr>
<tr>
<td>Windows</td>
<td>&lt;Ctrl&gt; z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Common Programming Error 11.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using the wrong file pointer to refer to a file is a logic error.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error-Prevention Tip 11.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Be sure that calls to file processing functions in a program contain the correct file pointers.</td>
</tr>
</tbody>
</table>

After the user enters end-of-file, the program closes the clients.dat file with fclose and terminates. Function fclose also receives the file pointer (rather than the file name) as an argument. If function fclose is not called explicitly, the operating system normally will close the file when program execution terminates. This is an example of operating system “housekeeping.”

<table>
<thead>
<tr>
<th>Good Programming Practice 11.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicitly close each file as soon as it’s no longer needed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance Tip 11.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closing a file can free resources for which other users or programs may be waiting.</td>
</tr>
</tbody>
</table>

In the sample execution for the program of Fig. 11.3, the user enters information for five accounts, then enters end-of-file to signal that data entry is complete. The sample execution does not show how the data records actually appear in the file. To verify that the file has been created successfully, in the next section we present a program that reads the file and prints its contents.

Figure 11.5 illustrates the relationship between FILE pointers, FILE structures and FCBs in memory. When the file "clients.dat" is opened, an FCB for the file is copied
Chapter 11  C File Processing

User has access to this

cfPtr = fopen( "clients.dat", "w" );

fopen returns a pointer to a FILE structure (defined in <stdio.h>).

Only the operating system has access to this

Open File Table

FCB for "clients.dat"

This entry is copied from FCB on disk when the file is opened.

When the program issues an I/O call such as

fprintf( cfPtr, "%d %s %.2f", account, name, balance );

the program locates the descriptor (7) in the FILE structure and uses the descriptor to find the FCB in the Open File Table.

Fig. 11.5  Relationship between FILE pointers, FILE structures and FCBs.

into memory. The figure shows the connection between the file pointer returned by fopen and the FCB used by the operating system to administer the file.

Programs may process no files, one file or several files. Each file used in a program must have a unique name and will have a different file pointer returned by fopen. All subsequent file processing functions after the file is opened must refer to the file with the
appropriate file pointer. Files may be opened in one of several modes (Fig. 11.6). To create a file, or to discard the contents of a file before writing data, open the file for writing ("w"). To read an existing file, open it for reading ("r"). To add records to the end of an existing file, open the file for appending ("a"). To open a file so that it may be written to and read from, open the file for updating in one of the three update modes—"r+", "w+" or "a+". Mode "r+" opens a file for reading and writing. Mode "w+" creates a file for reading and writing. If the file already exists, the file is opened and the current contents of the file are discarded. Mode "a+" opens a file for reading and writing—all writing is done at the end of the file. If the file does not exist, it’s created. Each file open mode has a corresponding binary mode (containing the letter b) for manipulating binary files. The binary modes are used in Sections 11.6–11.10 when we introduce random-access files. If an error occurs while opening a file in any mode, fopen returns NULL.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Open an existing file for reading.</td>
</tr>
<tr>
<td>w</td>
<td>Create a file for writing. If the file already exists, discard the current contents.</td>
</tr>
<tr>
<td>a</td>
<td>Append; open or create a file for writing at the end of the file.</td>
</tr>
<tr>
<td>r+</td>
<td>Open an existing file for update (reading and writing).</td>
</tr>
<tr>
<td>w+</td>
<td>Create a file for update. If the file already exists, discard the current contents.</td>
</tr>
<tr>
<td>a+</td>
<td>Append: open or create a file for update; writing is done at the end of the file.</td>
</tr>
<tr>
<td>rb</td>
<td>Open an existing file for reading in binary mode.</td>
</tr>
<tr>
<td>wb</td>
<td>Create a file for writing in binary mode. If the file already exists, discard the current contents.</td>
</tr>
<tr>
<td>ab</td>
<td>Append; open or create a file for writing at the end of the file in binary mode.</td>
</tr>
<tr>
<td>rb+</td>
<td>Open an existing file for update (reading and writing) in binary mode.</td>
</tr>
<tr>
<td>wb+</td>
<td>Create a file for update in binary mode. If the file already exists, discard the current contents.</td>
</tr>
<tr>
<td>ab+</td>
<td>Append: open or create a file for update in binary mode; writing is done at the end of the file.</td>
</tr>
</tbody>
</table>

Fig. 11.6  | File opening modes.

**Common Programming Error 11.4**
Opening a nonexistent file for reading is an error.

**Common Programming Error 11.5**
Opening a file for reading or writing without having been granted the appropriate access rights to the file (this is operating-system dependent) is an error.

**Common Programming Error 11.6**
Opening a file for writing when no disk space is available is an error.
11.5 Reading Data from a Sequential-Access File

Data is stored in files so that the data can be retrieved for processing when needed. The previous section demonstrated how to create a file for sequential access. This section shows how to read data sequentially from a file.

Figure 11.7 reads records from the file "clients.dat" created by the program of Fig. 11.3 and prints the contents of the records. Line 11 indicates that cfPtr is a pointer to a FILE. Line 14 attempts to open the file "clients.dat" for reading ("r") and determines whether the file is opened successfully (i.e., fopen does not return NULL). Line 19 reads a “record” from the file. Function fscanf is equivalent to function scanf, except fscanf receives as an argument a file pointer for the file from which the data is read. After this statement executes the first time, account will have the value 100, name will have the value "Jones" and balance will have the value 24.98. Each time the second fscanf statement (line 24) executes, the program reads another record from the file and account, name and balance take on new values. When the program reaches the end of the file, the file is closed (line 27) and the program terminates. Functionfeof returns true only after the program attempts to read the nonexistent data following the last line.

```c
/* Fig. 11.7: fig11_07.c
Reading and printing a sequential file */
#include <stdio.h>

int main( void )
{
    int account;    /* account number */
    char name[ 30 ]; /* account name */
    double balance; /* account balance */

    FILE *cfPtr;     /* cfPtr = clients.dat file pointer */

    /* fopen opens file; exits program if file cannot be opened */
    if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
        printf( "File could not be opened\n" );
    } /* end if */
    else { /* read account, name and balance from file */
        printf( "%-10s%-13s%11lf\n", "Account", "Name", "Balance" );
        fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
    }
}
```

Common Programming Error 11.7
Opening a file with the incorrect file mode is a logic error. For example, opening a file in write mode ("w") when it should be opened in update mode ("r+") causes the contents of the file to be discarded.

Error-Prevention Tip 11.2
Open a file only for reading (and not update) if the contents of the file should not be modified. This prevents unintentional modification of the file’s contents. This is another example of the principle of least privilege.
Resetting the File Position Pointer

To retrieve data sequentially from a file, a program normally starts reading from the beginning of the file and reads all data consecutively until the desired data is found. It may be desirable to process the data sequentially in a file several times (from the beginning of the file) during the execution of a program. A statement such as

```
rewind(cfPtr);
```

causes a program’s file position pointer—which indicates the number of the next byte in the file to be read or written—to be repositioned to the beginning of the file (i.e., byte 0) pointed to by cfPtr. The file position pointer is not really a pointer. Rather it’s an integer value that specifies the byte location in the file at which the next read or write is to occur. This is sometimes referred to as the file offset. The file position pointer is a member of the FILE structure associated with each file.

Credit Inquiry Program

The program of Fig. 11.8 allows a credit manager to obtain lists of customers with zero balances (i.e., customers who do not owe any money), customers with credit balances (i.e., customers to whom the company owes money) and customers with debit balances (i.e., customers who owe the company money for goods and services received). A credit balance is a negative amount; a debit balance is a positive amount.

The program displays a menu and allows the credit manager to enter one of three options to obtain credit information. Option 1 produces a list of accounts with zero balances. Option 2 produces a list of accounts with credit balances. Option 3 produces a list of accounts with debit balances. Option 4 terminates program execution. A sample output is shown in Fig. 11.9.
/* Fig. 11.8: fig11_08.c */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    int request;  /* request number */
    int account;  /* account number */
    double balance; /* account balance */
    char name[ 30 ]; /* account name */
    FILE *cfPtr;  /* clients.dat file pointer */

    /* fopen opens the file; exits program if file cannot be opened */
    if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
        printf( "File could not be opened\n" );
    } /* end if */
    else {

        /* display request options */
        printf( "Enter request\n"
            " 1 - List accounts with zero balances\n"            " 2 - List accounts with credit balances\n"            " 3 - List accounts with debit balances\n"            " 4 - End of run\n? " );
        scanf( "%d", &request );

        /* process user's request */
        while ( request != 4 ) {

            /* read account, name and balance from file */
            fscanf( cfPtr, "%d%s%lf", &account, name, &balance );

            switch ( request ) {

                case 1:
                    printf( "\nAccounts with zero balances:\n" );
                    /* read file contents (until eof) */
                    while ( !feof( cfPtr ) ) {
                        if ( balance == 0 ) {
                            printf( "%-10d%-13s%7.2f\n", account, name, balance );
                        } /* end if */
                    } /* end while */

                case 2:
                    printf( "\nAccounts with credit balances:\n" );

                    /* read account, name and balance from file */
                    fscanf( cfPtr, "%d%s%lf", &account, name, &balance );

                    /* read file contents (until eof) */
                    while ( !feof( cfPtr ) ) {
                        if ( balance == 0 ) {
                            printf( "%-10d%-13s%7.2f\n", account, name, balance );
                        } /* end if */
                    } /* end while */

                    break;

            } /* end switch */

        } /* end while */

    } /* end else */
}

Fig. 11.8  |  Credit inquiry program. (Part 1 of 2.)
11.5 Reading Data from a Sequential-Access File

Data in this type of sequential file cannot be modified without the risk of destroying other data. For example, if the name “White” needed to be changed to “Worthington,” the old name cannot simply be overwritten. The record for White was written to the file as

```
300 White 0.00
```

Fig. 11.8  Credit inquiry program. (Part 2 of 2.)
If the record is rewritten beginning at the same location in the file using the new name, the record would be

300 Worthington 0.00

The new record is larger (has more characters) than the original record. The characters beyond the second “o” in “Worthington” would overwrite the beginning of the next sequential record in the file. The problem here is that in the formatted input/output model using `fprintf` and `fscanf`, fields—and hence records—can vary in size. For example, the values 7, 14, −117, 2074 and 27383 are all ints stored in the same number of bytes internally, but they are different-sized fields when displayed on the screen or written to a file as text.

Therefore, sequential access with `fprintf` and `fscanf` is not usually used to update records in place. Instead, the entire file is usually rewritten. To make the preceding name change, the records before 300 White 0.00 in such a sequential-access file would be copied to a new file, the new record would be written and the records after 300 White 0.00 would be copied to the new file. This requires processing every record in the file to update one record.

### 11.6 Random-Access Files

As we stated previously, records in a file created with the formatted output function `fprintf` are not necessarily the same length. However, individual records of a random-access file are normally fixed in length and may be accessed directly (and thus quickly) without searching through other records. This makes random-access files appropriate for airline reservation systems, banking systems, point-of-sale systems, and other kinds of
transaction processing systems that require rapid access to specific data. There are other ways of implementing random-access files, but we’ll limit our discussion to this straightforward approach using fixed-length records.

Because every record in a random-access file normally has the same length, the exact location of a record relative to the beginning of the file can be calculated as a function of the record key. We’ll soon see how this facilitates immediate access to specific records, even in large files.

Figure 11.10 illustrates one way to implement a random-access file. Such a file is like a freight train with many cars—some empty and some with cargo. Each car in the train is the same length.

![Fig. 11.10 | C’s view of a random-access file.](image)

Fixed-length records enable data to be inserted in a random-access file without destroying other data in the file. Data stored previously can also be updated or deleted without rewriting the entire file. In the following sections we explain how to create a random-access file, enter data, read the data both sequentially and randomly, update the data, and delete data no longer needed.

### 11.7 Creating a Random-Access File

Function `fwrite` transfers a specified number of bytes beginning at a specified location in memory to a file. The data is written beginning at the location in the file indicated by the file position pointer. Function `fread` transfers a specified number of bytes from the location in the file specified by the file position pointer to an area in memory beginning with a specified address. Now, when writing an integer, instead of using

```c
fprintf( fPtr, "%d", number );
```

which could print a single digit or as many as 11 digits (10 digits plus a sign, each of which requires 1 byte of storage) for a 4-byte integer, we can use

```c
fwrite( &number, sizeof( int ), 1, fPtr );
```

which always writes 4 bytes (or 2 bytes on a system with 2-byte integers) from a variable `number` to the file represented by `fPtr` (we’ll explain the 1 argument shortly). Later, `fread` can be used to read 4 of those bytes into an integer variable `number`. Although `fread` and `fwrite` read and write data, such as integers, in fixed-size rather than variable-size format,
the data they handle are processed in computer “raw data” format (i.e., bytes of data) rather than in printf’s and scanf’s human-readable text format. Since the “raw” representation of data is system-dependent, “raw data” may not be readable on other systems, or by programs produced by other compilers or with other compiler options.

Functions fwrite and fread are capable of reading and writing arrays of data to and from disk. The third argument of both fread and fwrite is the number of elements in the array that should be read from disk or written to disk. The preceding fwrite function call writes a single integer to disk, so the third argument is 1 (as if one element of an array is being written).

File processing programs rarely write a single field to a file. Normally, they write one struct at a time, as we show in the following examples.

Consider the following problem statement:

Create a credit processing system capable of storing up to 100 fixed-length records. Each record should consist of an account number that will be used as the record key, a last name, a first name and a balance. The resulting program should be able to update an account, insert a new account record, delete an account and list all the account records in a formatted text file for printing. Use a random-access file.

The next several sections introduce the techniques necessary to create the credit processing program. Figure 11.11 shows how to open a random-access file, define a record format using a struct, write data to the disk and close the file. This program initializes all 100 records of the file “credit.dat” with empty structs using the function fwrite. Each empty struct contains 0 for the account number, “” (the empty string) for the last name, “” for the first name and 0.0 for the balance. The file is initialized in this manner to create space on the disk in which the file will be stored and to make it possible to determine if a record contains data.

```c
/* Fig. 11.11: fig11_11.c
   Creating a random-access file sequentially */
#include <stdio.h>

/* clientData structure definition */
struct clientData {
  int acctNum; /* account number */
  char lastName[15]; /* account last name */
  char firstName[10]; /* account first name */
  double balance; /* account balance */
}; /* end structure clientData */

int main( void )
{
  int i; /* counter used to count from 1-100 */
  /* create clientData with default information */
  struct clientData blankClient = { 0, "", "", 0.0 };
  FILE *cfPtr; /* credit.dat file pointer */
```

Fig. 11.11 | Creating a random access file sequentially. (Part 1 of 2.)
11.8 Writing Data Randomly to a Random-Access File

Function fwrite writes a block (specific number of bytes) of data to a file. In our program, line 29 causes the structure blankClient of size sizeof(struct clientData) to be written to the file pointed to by cfPtr. The operator sizeof returns the size in bytes of its operand in parentheses (in this case struct clientData). The sizeof operator returns an unsigned integer and can be used to determine the size in bytes of any data type or expression. For example, sizeof(int) can be used to determine whether an integer is stored in 2 or 4 bytes on a particular computer.

Function fwrite can actually be used to write several elements of an array of objects. To write several array elements, supply in the call to fwrite a pointer to an array as the first argument and the number of elements to be written as the third argument. In the preceding statement, fwrite was used to write a single object that was not an array element. Writing a single object is equivalent to writing one element of an array, hence the 1 in the fwrite call. [Note: Figures 11.12, 11.15 and 11.16 use the data file created in Fig. 11.11, so you must run Fig. 11.11 before Figs. 11.12, 11.15 and 11.16]

### 11.8 Writing Data Randomly to a Random-Access File

Figure 11.12 writes data to the file "credit.dat". It uses the combination of fseek and fwrite to store data at specific locations in the file. Function fseek sets the file position pointer to a specific position in the file, then fwrite writes the data. A sample execution is shown in Fig. 11.13.
Chapter 11  C File Processing

Lines 40–41 position the file position pointer for the file referenced by *cfPtr to the byte location calculated by 

\((\text{client.accountNum} - 1) \times \text{sizeof(\text{struct clientData})}\).

The value of this expression is called the offset or the displacement. Because the account number is between 1 and 100 but the byte positions in the file start with 0, 1 is subtracted

```
int main( void )
{
    FILE *cfPtr; /* credit.dat file pointer */

    /* create clientData with default information */
    struct clientData client = { 0, "", "", 0.0};

    /* fopen opens the file; exits if file cannot be opened */
    if (( cfPtr = fopen("credit.dat", "rb+")) == NULL ) {
        printf( "File could not be opened.\n" );
    } /* end if */
    else {
        /* require user to specify account number */
        printf( "Enter account number
" );
        scanf("%d", &client.acctNum);

        /* user enters information, which is copied into file */
        while ( client.acctNum != 0 ) {
            /* user enters last name, first name and balance */
            printf( "Enter lastname, firstname, balance\n? " );
            set record lastName, firstName and balance value */
            fscanf( stdin, "%s%s%lf", client.lastName,
                        client.firstName, &client.balance );

            /* seek position in file to user-specified record */
            fseek( cfPtr, ( client.acctNum - 1 ) *
                    sizeof( struct clientData ), SEEK_SET );

            /* write user-specified information in file */
            fwrite( &client, sizeof( struct clientData ), 1, cfPtr );

            /* enable user to input another account number */
            printf( "Enter account number\n? " );
            scanf("%d", &client.acctNum);
        } /* end while */

        fclose( cfPtr ); /* fclose closes the file */
    } /* end else */

    return 0; /* indicates successful termination */
} /* end main */
```

Fig. 11.12  Writing data randomly to a random-access file. (Part 2 of 2.)
from the account number when calculating the byte location of the record. Thus, for record 1, the file position pointer is set to byte 0 of the file. The symbolic constant SEEK_SET indicates that the file position pointer is positioned relative to the beginning of the file by the amount of the offset. As the above statement indicates, a seek for account number 1 in the file sets the file position pointer to the beginning of the file because the byte location calculated is 0. Figure 11.14 illustrates the file pointer referring to a FILE structure in memory. The file position pointer in this diagram indicates that the next byte to be read or written is 5 bytes from the beginning of the file.
The function prototype for _fseek_ is

\[
\text{int } \text{fseek}( \text{FILE } *\text{stream}, \text{long int } \text{offset}, \text{int } \text{whence} );
\]

where _offset_ is the number of bytes to seek from location _whence_ in the file pointed to by _stream_. The argument _whence_ can have one of three values—SEEK_SET, SEEK_CUR or SEEK_END (all defined in `<stdio.h>`)—indicating the location in the file from which the seek begins. SEEK_SET indicates that the seek starts at the beginning of the file; SEEK_CUR indicates that the seek starts at the current location in the file; and SEEK_END indicates that the seek starts at the end of the file.

For simplicity, the programs in this chapter do not perform error checking. If you wish to determine whether functions like _fscanf_ (lines 36–37), _fseek_ (lines 40–41) and _fwrite_ (line 44) operate correctly, you can check their return values. Function _fscanf_ returns the number of data items successfully read or the value EOF if a problem occurs while reading data. Function _fseek_ returns a nonzero value if the seek operation cannot be performed. Function _fwrite_ returns the number of items it successfully output. If this number is less than the third argument in the function call, then a write error occurred.

### 11.9 Reading Data from a Random-Access File

Function _fread_ reads a specified number of bytes from a file into memory. For example,

\[
\text{fread( } \&\text{client}, \text{sizeof( struct clientData ), 1, cfPtr );}
\]

reads the number of bytes determined by `sizeof(struct clientData)` from the file referenced by _cfPtr_ and stores the data in the structure _client_. The bytes are read from the location in the file specified by the file position pointer. Function _fread_ can be used to read several fixed-size array elements by providing a pointer to the array in which the elements will be stored and by indicating the number of elements to be read. The preceding statement specifies that one element should be read. To read more than one element, specify the number of elements in the third argument of the _fread_ statement. Function _fread_ returns the number of items it successfully input. If this number is less than the third argument in the function call, then a read error occurred.

Figure 11.15 reads sequentially every record in the "credit.dat" file, determines whether each record contains data and displays the formatted data for records containing data. Function _feof_ determines when the end of the file is reached, and the _fread_ function transfers data from the disk to the _clientData_ structure _client_.
11.10 Case Study: Transaction-Processing Program

We now present a substantial transaction-processing program using random-access files. The program maintains a bank’s account information. The program updates existing accounts, adds new accounts, deletes accounts and stores a listing of all the current accounts in a text file for printing. We assume that the program of Fig. 11.11 has been executed to create the file credit.dat.

The program has five options. Option 1 calls function textFile (lines 65–95) to store a formatted list of all the accounts in a text file called accounts.txt that may be printed.
later. The function uses fread and the sequential file access techniques used in the program of Fig. 11.15. After choosing option 1 the file accounts.txt contains:

<table>
<thead>
<tr>
<th>Acct</th>
<th>Last Name</th>
<th>First Name</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>Brown</td>
<td>Nancy</td>
<td>-24.54</td>
</tr>
<tr>
<td>33</td>
<td>Dunn</td>
<td>Stacey</td>
<td>314.33</td>
</tr>
<tr>
<td>37</td>
<td>Barker</td>
<td>Doug</td>
<td>0.00</td>
</tr>
<tr>
<td>88</td>
<td>Smith</td>
<td>Dave</td>
<td>258.34</td>
</tr>
<tr>
<td>96</td>
<td>Stone</td>
<td>Sam</td>
<td>34.98</td>
</tr>
</tbody>
</table>

Option 2 calls the function updateRecord (lines 98–142) to update an account. The function will only update a record that already exists, so the function first checks to see if the record specified by the user is empty. The record is read into structure client with fread, then member acctNum is compared to 0. If it’s 0, the record contains no information, and a message is printed stating that the record is empty. Then, the menu choices are displayed. If the record contains information, function updateRecord inputs the transaction amount, calculates the new balance and rewrites the record to the file. A typical output for option 2 is

```
Enter account to update ( 1 - 100 ): 37
37 Barker Doug 0.00
Enter charge ( + ) or payment ( - ): +87.99
37 Barker Doug 87.99
```

Option 3 calls the function newRecord (lines 179–218) to add a new account to the file. If the user enters an account number for an existing account, newRecord displays an error message that the record already contains information, and the menu choices are printed again. This function uses the same process to add a new account as does the program in Fig. 11.12. A typical output for option 3 is

```
Enter new account number ( 1 - 100 ): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

Option 4 calls function deleteRecord (lines 145–176) to delete a record from the file. Deletion is accomplished by asking the user for the account number and reinitializing the record. If the account contains no information, deleteRecord displays an error message that the account does not exist. Option 5 terminates program execution. The program is shown in Fig. 11.16. The file "credit.dat" is opened for update (reading and writing) using "rb+" mode.

```
/* Fig. 11.16: fig11_16.c */
This program reads a random access file sequentially, updates data
already written to the file, creates new data to be placed in the
file, and deletes data previously in the file.
```

---

**Fig. 11.16** Bank account program. (Part 1 of 6.)
```c
#include <stdio.h>

/* prototypes */
int enterChoice( void );
void textFile( FILE *readPtr );
void updateRecord( FILE *fPtr );
void newRecord( FILE *fPtr );
void deleteRecord( FILE *fPtr );

int main( void )
{
    FILE *cfPtr; /* credit.dat file pointer */
    int choice; /* user's choice */

    /* fopen opens the file; exits if file cannot be opened */
    if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL ) {
        printf( "File could not be opened.\n" );
    } /* end if */
    else {
        /* enable user to specify action */
        while ( ( choice = enterChoice() ) != 5 ) {
            switch ( choice ) {
                case 1:
                    textFile( cfPtr );
                    break;
                /* update record */
                case 2:
                    updateRecord( cfPtr );
                    break;
                /* create record */
                case 3:
                    newRecord( cfPtr );
                    break;
                /* delete existing record */
                case 4:
                    deleteRecord( cfPtr );
                    break;
                /* display message if user does not select valid choice */
                default:
                    printf( "Incorrect choice\n" );
                    break;
            } /* end switch */
        } /* end while */
    } /* end else */
}
```

---

**Fig. 11.16** | Bank account program. (Part 2 of 6.)
fclose( cfPtr ); /* fclose closes the file */
} /* end else */
return 0; /* indicates successful termination */
} /* end main */

/* create formatted text file for printing */
void textFile( FILE *readPtr )
{
    FILE *writePtr; /* accounts.txt file pointer */

    /* create clientData with default information */
    struct clientData client = { 0, "", "", 0.0 };;

    /* fopen opens the file; exits if file cannot be opened */
    if ( ( writePtr = fopen( "accounts.txt", "w" ) ) == NULL ) {
        printf( "File could not be opened.\n" );
    } /* end if */
    else {
        rewind( readPtr ); /* sets pointer to beginning of file */
        fprintf( writePtr, "%-6s%-16s%-11s%10s\n", "Acct", "Last Name", "First Name","Balance" );

        /* copy all records from random-access file into text file */
        while ( !feof( readPtr ) ) {
            fread( &client, sizeof( struct clientData ), 1, readPtr );

            /* write single record to text file */
            if ( client.acctNum != 0 ) {
                fprintf( writePtr, "%-6d%-16s%-11s%10.2f\n", client.acctNum, client.lastName, client.firstName, client.balance );
            } /* end if */
        } /* end while */
    } /* end else */
fclose( writePtr ); /* fclose closes the file */
} /* end function textFile */

/* update balance in record */
void updateRecord( FILE *fPtr )
{
    int account;        /* account number */
    double transaction; /* transaction amount */

    /* create clientData with no information */
    struct clientData client = { 0, "", "", 0.0 };;

    /* obtain number of account to update */
    printf( "Enter account to update ( 1 - 100 ): " );
    scanf( "%d", &account );

Fig. 11.16 | Bank account program. (Part 3 of 6.)
/* move file pointer to correct record in file */
fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ), SEEK_SET );

/* read record from file */
fwrite( &client, sizeof( struct clientData ), 1, fPtr );

/* display error if account does not exist */
if ( client.acctNum == 0 ) {
    printf("Account #%d has no information.\n", account );
} /* end if */
else { /* update record */
    printf("%-6d%-16s%-11s%10.2f\n", account );
    scanf("%ld", &transaction );
    client.balance += transaction; /* update record balance */
    printf("%-6d%-16s%-11s%10.2f\n", account );
} /* end else */
} /* end function updateRecord */

/* delete an existing record */
void deleteRecord( FILE *fPtr )
{
    struct clientData client; /* stores record read from file */
    struct clientData blankClient = { 0, "", "", 0 }; /* blank client */
    int accountNum; /* account number */

    /* obtain number of account to delete */
    printf("Enter account number to delete ( 1 - 100 ): ");
    scanf("%d", &accountNum );

    /* move file pointer to correct record in file */
    fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ), SEEK_SET );

    /* read record from file */
    fread( &client, sizeof( struct clientData ), 1, fPtr );

    /* move file pointer to correct record in file */
    fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ), SEEK_SET );

    /* write updated record over old record in file */
    fwrite( &client, sizeof( struct clientData ), 1, fPtr );
} /* end function updateRecord */

/* delete an existing record */
void deleteRecord( FILE *fPtr )
{
/* display error if record does not exist */
if ( client.acctNum == 0 ) {
    printf( "Account %d does not exist.\n", accountNum );
} /* end if */
else { /* delete record */
    /* move file pointer to correct record in file */
    if ( client.acctNum != 0 ) {
        printf( "Account #%d already contains information.\n", client.acctNum );
    } /* end if */
    else { /* create record */
        /* move file pointer to correct record in file */
        fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ), SEEK_SET );

        /* replace existing record with blank record */
        fwrite( &blankClient, sizeof( struct clientData ), 1, fPtr );
    } /* end else */
} /* end function deleteRecord */

/* create and insert record */
void newRecord( FILE *fPtr )
{
    /* create clientData with default information */
    struct clientData client = { 0, "", "", 0.0 };

    int accountNum; /* account number */

    /* obtain number of account to create */
    printf( "Enter new account number ( 1 - 100 ): " );
    scanf( "%d", &accountNum );

    /* move file pointer to correct record in file */
    fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ), SEEK_SET );

    /* read record from file */
    fread( &client, sizeof( struct clientData ), 1, fPtr );

    /* display error if account already exists */
    if ( client.acctNum != 0 ) {
        printf( "Account #%d already contains information.\n", client.acctNum );
    } /* end if */
    else { /* create record */
        /* user enters last name, first name and balance */
        printf( "Enter lastname, firstname, balance\n? " );
        scanf( "%s%s%lf", &client.lastName, &client.firstName, &client.balance );

        client.acctNum = accountNum;

        /* move file pointer to correct record in file */
        fseek( fPtr, ( client.acctNum - 1 ) * sizeof( struct clientData ), SEEK_SET );

        /* move file pointer to correct record in file */
        fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ), SEEK_SET );

        /* read record from file */
        fread( &client, sizeof( struct clientData ), 1, fPtr );

        /* display error if account already exists */
        if ( client.acctNum != 0 ) {
            printf( "Account #%d already contains information.\n", client.acctNum );
        } /* end if */
        else { /* create record */
            /* user enters last name, first name and balance */
            printf( "Enter lastname, firstname, balance\n? " );
            scanf( "%s%s%lf", &client.lastName, &client.firstName, &client.balance );

            client.acctNum = accountNum;

            /* move file pointer to correct record in file */
            fseek( fPtr, ( client.acctNum - 1 ) * sizeof( struct clientData ), SEEK_SET );

        } /* end else */
    } /* end else */
} /* end function newRecord */
Summary

Section 11.1 Introduction
• Files are used for permanent retention of large amounts of data.
• Computers store files on secondary storage devices, especially disk storage devices.

Section 11.2 Data Hierarchy
• The smallest data item in a computer can assume the value 0 or the value 1. Such a data item is called a bit (short for “binary digit”—a digit that can assume one of two values).
• Computer circuitry performs various simple bit manipulations such as determining a bit’s value, setting a bit’s value and reversing a bit (from 1 to 0 or from 0 to 1).
• Programmers prefer to work with data in the form of decimal digits, letters and special symbols, which are referred to as characters.
• The set of all characters that may be used to write programs and represent data items on a particular computer is called that computer’s character set.
• Every character in a computer’s character set is represented as a pattern of 1s and 0s (called a byte).
• Bytes are most commonly composed of eight bits.
• Fields are composed of characters. A field is a group of characters that conveys meaning.
• A record (i.e., a struct) is a group of related fields.
• A file is a group of related records.
• To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a record key. A record key identifies a record as belonging to a particular person or entity.
Chapter 11  C File Processing

- The most popular type of file organization is called a sequential file, in which records are typically stored in order by the record key field.

Section 11.3 Files and Streams
- C views each file as a sequential stream of bytes. When a file is opened, a stream is associated with the file.
- Three files and their associated streams are automatically opened when program execution begins—the standard input, the standard output and the standard error.
- Streams provide communication channels between files and programs.
- The standard input stream enables a program to read data from the keyboard, and the standard output stream enables a program to print data on the screen.
- Opening a file returns a pointer to a FILE structure (defined in <stdio.h>) that contains information used to process the file. This structure includes a file descriptor, i.e., an index into an operating system array called the open file table. Each array element contains a file control block (FCB) that the operating system uses to administer a particular file.
- The standard input, standard output and standard error are manipulated using file pointers stdin, stdout and stderr.
- Function fgetc reads one character from a file. It receives as an argument a FILE pointer for the file from which a character will be read.
- Function fputc writes one character to a file. It receives as arguments a character to be written and a pointer for the file to which the character will be written.
- The fgets and fputs functions read a line from a file or write a line to a file, respectively.

Section 11.4 Creating a Sequential-Access File
- C imposes no structure on a file. You must provide a file structure to meet the requirements of a particular application.
- A C program administers each file with a separate FILE structure.
- Each open file must have a separately declared pointer of type FILE that is used to refer to the file.
- Function fopen takes as arguments a file name and a file open mode and returns a pointer to the FILE structure for the file opened.
- The file open mode "w" indicates that the file is to be opened for writing. If the file does not exist, fopen creates the file. If the exists, the contents are discarded without warning.
- Function fopen returns NULL if it’s unable to open a file.
- Functionfeof receives a pointer to a FILE and returns a nonzero (true) value when the end-of-file indicator has been set; otherwise, the function returns zero.
- Function fprintf is equivalent to printf except that fprintf also receives as an argument a file pointer for the file to which the data will be written.
- Function fclose receives a file pointer as an argument and closes the specified file.
- When a file is opened, the file control block (FCB) for the file is copied into memory. The FCB is used by the operating system to administer the file.
- To create a file, or to discard the file’s contents before writing data, open the file for writing ("w").
- To read an existing file, open it for reading ("r").
- To add records to the end of an existing file, open the file for appending ("a").
- To open a file so that it may be written to and read from, open the file for updating in one of the three update modes—"r+", "w+" or "a+". Mode "r+" opens a file for reading and writing. Mode
"w+" creates a file for reading and writing. If the file already exists, it’s opened and its contents are discarded. Mode "a+" opens a file for reading and writing—all writing is done at the end of the file. If the file does not exist, it’s created.

• Each file open mode has a corresponding binary mode (containing the letter b) for manipulating binary files.

Section 11.5 Reading Data from a Sequential-Access File

• Function `fscanf` is equivalent to function `scanf` except `fscanf` receives as an argument a file pointer for the file from which the data is read.

• To retrieve data sequentially from a file, a program normally starts reading from the beginning of the file and reads all data consecutively until the desired data is found.

• Function `rewind` causes a program’s file position pointer to be repositioned to the beginning of the file (i.e., byte 0) pointed to its argument.

• The file position pointer is an integer value that specifies the byte location in the file at which the next read or write is to occur. This is sometimes referred to as the file offset. The file position pointer is a member of the `FILE` structure associated with each file.

• The data in a sequential file typically cannot be modified without the risk of destroying other data in the file.

Section 11.6 Random-Access Files

• Individual records of a random-access file are normally fixed in length and may be accessed directly (and thus quickly) without searching through other records.

• Because every record in a random-access file normally has the same length, the exact location of a record relative to the beginning of the file can be calculated as a function of the record key.

• Fixed-length records enable data to be inserted in a random-access file without destroying other data. Data stored previously can also be updated or deleted without rewriting the entire file.

Section 11.7 Creating a Random-Access File

• Function `fwrite` transfers a specified number of bytes beginning at a specified location in memory to a file. The data is written beginning at the file position pointer’s location.

• Function `fread` transfers a specified number of bytes from the location in the file specified by the file position pointer to an area in memory beginning with a specified address.

• Functions `fwrite` and `fread` are capable of reading and writing arrays of data from and to disk. The third argument of both `fread` and `fwrite` is the number of elements to process in the array.

• File processing programs normally write one `struct` at a time.

• Function `fwrite` writes a block (specific number of bytes) of data to a file.

• To write several array elements, supply in the call to `fwrite` a pointer to an array as the first argument and the number of elements to be written as the third argument.

Section 11.8 Writing Data Randomly to a Random-Access File

• Function `fseek` sets the file position pointer for a given file to a specific position in the file. Its second argument indicates the number of bytes to seek and its third argument indicates the location from which to seek. The third argument can have one of three values—SEEK_SET, SEEK_CUR or SEEK_END (all defined in `<stdio.h>`). SEEK_SET indicates that the seek starts at the beginning of the file; SEEK_CUR indicates that the seek starts at the current location in the file; and SEEK_END indicates that the seek starts at the end of the file.

• If you wish to determine whether functions like `fscanf`, `fseek` and `fwrite` operate correctly, you can check their return values.
• Function `fscanf` returns the number of fields successfully read or the value `EOF` if a problem occurs while reading data.
• Function `fseek` returns a nonzero value if the seek operation cannot be performed.
• Function `fwrite` returns the number of items it successfully output. If this number is less than the third argument in the function call, then a write error occurred.

Section 11.9 Reading Data from a Random-Access File
• Function `fread` reads a specified number of bytes from a file into memory.
• Function `fread` can be used to read several fixed-size array elements by providing a pointer to the array in which the elements will be stored and by indicating the number of elements to be read.
• Function `fread` returns the number of items it successfully input. If this number is less than the third argument in the function call, then a read error occurred.

Terminology

binary digit 418
bit 418
byte 418
character 418
character set 418
data hierarchy 418
database 420
database management system (DBMS) 420
end-of-file marker 420
fclose function 423
feof function 422
fgetc function 420
fgets function 420
fread function 420
fread function 421
fscanf function 421
fseek function 423
fwrite function 421
getchar function 420
letter 418
NULL 422
offset 434
open file table 420
printf function 421
putchar function 420
random-access file 430
record 419
record key 419
SEEK_CUR 436
SEEK_END 436
SEEK_SET 435
sequential file 420
special symbol 418
standard error 420
standard input file 420
standard output file 420
stderr (the standard error device) 420
stdin (the standard input device) 420
stdout (the standard output device) 420
stream 420
transaction-processing system 431
zeros and ones 418

Self-Review Exercises

11.1 Fill in the blanks in each of the following:
a) Ultimately, all data items processed by a computer are reduced to combinations of ________ and ________.
b) The smallest data item a computer can process is called a(n) ________.
c) A(n) ________ is a group of related records.
d) Digits, letters and special symbols are referred to as ________. 
Self-Review Exercises

447

e) A group of related files is called a _______.
f) Function _______ closes a file.
g) The ______ function reads data from a file in a manner similar to how scanf reads from stdin.
h) Function _______ reads a character from a specified file.
i) Function _______ reads a line from a specified file.
j) Function _______ opens a file.
k) Function _______ is normally used when reading data from a file in random-access applications.
l) Function _______ repositions the file position pointer to a specific location in the file.

11.2 State which of the following are true and which are false. If false, explain why.
a) Function fscanf cannot be used to read data from the standard input.
b) You must explicitly use fopen to open the standard input, standard output and standard error streams.
c) A program must explicitly call function fclose to close a file.
d) If the file position pointer points to a location in a sequential file other than the beginning of the file, the file must be closed and reopened to read from the beginning of the file.
e) Function fprintf can write to the standard output.
f) Data in sequential-access files are always updated without overwriting other data.
g) It's not necessary to search through all the records in a random-access file to find a specific record.
h) Records in random-access files are not of uniform length.
i) Function fseek may only seek relative to the beginning of a file.

11.3 Write a single statement to accomplish each of the following. Assume that each of these statements applies to the same program.

a) Write a statement that opens the file "oldmast.dat" for reading and assigns the returned file pointer to ofPtr.
b) Write a statement that opens the file "trans.dat" for reading and assigns the returned file pointer to tfPtr.
c) Write a statement that opens the file "newmast.dat" for writing (and creation) and assigns the returned file pointer to nfPtr.
d) Write a statement that reads a record from the file "oldmast.dat". The record consists of integer accountNum, string name and floating-point currentBalance.
e) Write a statement that reads a record from the file "trans.dat". The record consists of the integer accountNum and floating-point dollarAmount.
f) Write a statement that writes a record to the file "newmast.dat". The record consists of the integer accountNum, string name and floating-point currentBalance.

11.4 Find the error in each of the following program segments and explain how to correct it.

a) The file referred to by fPtr ("payables.dat") has not been opened.
   printf( fPtr, "%d%s%d\n", account, company, amount );
b) open( "receive.dat", "r+" );
c) The following statement should read a record from the file "payables.dat". File pointer payPtr refers to this file, and file pointer recPtr refers to the file "receive.dat":
   scanf( recPtr, "%d%s%d\n", &account, company, &amount );
d) The file "tools.dat" should be opened to add data to the file without discarding the current data.
   if ( ( tfPtr = fopen( "tools.dat", "w" ) ) != NULL )
e) The file "courses.dat" should be opened for appending without modifying the current contents of the file.
   if ( ( cfPtr = fopen( "courses.dat", "w+" ) ) != NULL )
Answers to Self-Review Exercises

11.1  a) 1s, 0s.  b) Bit.  c) File.  d) Characters.  e) Database.  f) fclose.  g) fscanf.  h) fgetc.  i) fgets.  j) fopen.  k) fread.  l) fseek.

11.2  a) False. Function fscanf can be used to read from the standard input by including the pointer to the standard input stream, stdin, in the call to fscanf.

b) False. These three streams are opened automatically by C when program execution begins.

c) False. The files will be closed when program execution terminates, but all files should be explicitly closed with fclose.

d) False. Function rewind can be used to reposition the file position pointer to the beginning of the file.

e) True.

f) False. In most cases, sequential file records are not of uniform length. Therefore, it’s possible that updating a record will cause other data to be overwritten.

g) True.

h) False. Records in a random-access file are normally of uniform length.

i) False. It’s possible to seek from the beginning of the file, from the end of the file and from the current location in the file.

11.3  a) ofPtr = fopen("oldmast.dat", "r");

b) tfPtr = fopen("trans.dat", "r");

c) nfPtr = fopen("newmast.dat", "w");

d) fscanf( ofPtr, "%d%s%f", &accountNum, name, &currentBalance );

e) fscanf( tfPtr, "%d%f", &accountNum, &dollarAmount );

f) fprintf( nfPtr, "%d %s %.2f", accountNum, name, currentBalance );

11.4  a) Error: The file "payables.dat" has not been opened before the reference to its file pointer.

Correction: Use fopen to open "payables.dat" for writing, appending or updating.

b) Error: Function open is not a Standard C function.

Correction: Use function fopen.

c) Error: Function fscanf uses the incorrect file pointer to refer to file "payables.dat".

Correction: Use file pointer payPtr to refer to "payables.dat".

d) Error: The contents of the file are discarded because the file is opened for writing ("w").

Correction: To add data to the file, either open the file for updating ("r+") or open the file for appending ("a").

e) Error: File "courses.dat" is opened for updating in "w+" mode which discards the current contents of the file.

Correction: Open the file "a" mode.

Exercises

11.5  Fill in the blanks in each of the following:

a) Computers store large amounts of data on secondary storage devices as ________.

b) A(n) ________ is composed of several fields.

c) A field that may contain digits, letters and blanks is called a(n) ________ field.

d) To facilitate the retrieval of specific records from a file, one field in each record is chosen as a(n) ________.

e) Most information stored in computer systems is stored in ________ files.

f) A group of related characters that conveys meaning is called a(n) ________.
449

Exercises

g) The file pointers for the three files that are opened automatically when program execution begins are named _______, _______, and _______.

h) Function _______ writes a character to a specified file.

i) Function _______ writes a line to a specified file.

j) Function _______ is generally used to write data to a random-access file.

k) Function _______ repositions the file position pointer to the beginning of the file.

11.6 State which of the following are true and which are false. If false, explain why.

a) The impressive functions performed by computers essentially involve the manipulation of zeros and ones.

b) People prefer to manipulate bits instead of characters and fields because bits are more compact.

c) People specify programs and data items as characters; computers then manipulate and process these characters as groups of zeros and ones.

d) A person’s zip code is an example of a numeric field.

e) A person’s street address is generally considered to be an alphabetic field in computer applications.

f) Data items processed by a computer form a data hierarchy in which data items become larger and more complex as we progress from fields to characters to bits etc.

g) A record key identifies a record as belonging to a particular field.

h) Most companies store their information in a single file to facilitate computer processing.

i) Files are always referred to by name in C programs.

j) When a program creates a file, the file is automatically retained by the computer for future reference.

11.7 (File Matching) Exercise 11.3 asked the reader to write a series of single statements. Actually, these statements form the core of an important type of file-processing program, namely, a file-matching program. In commercial data processing, it’s common to have several files in each system. In an accounts receivable system, for example, there is generally a master file containing detailed information about each customer such as the customer’s name, address, telephone number, outstanding balance, credit limit, discount terms, contract arrangements and possibly a condensed history of recent purchases and cash payments.

As transactions occur (i.e., sales are made and cash payments arrive in the mail), they are entered into a file. At the end of each business period (i.e., a month for some companies, a week for others and a day in some cases) the file of transactions (called "trans.dat" in Exercise 11.3) is applied to the master file (called "oldmast.dat" in Exercise 11.3), thus updating each account’s record of purchases and payments. After each of these updatings run, the master file is rewritten as a new file ("newmast.dat"), which is then used at the end of the next business period to begin the updating process again.

File-matching programs must deal with certain problems that do not exist in single-file programs. For example, a match does not always occur. A customer on the master file might not have made any purchases or cash payments in the current business period, and therefore no record for this customer will appear on the transaction file. Similarly, a customer who did make some purchases or cash payments might have just moved to this community, and the company may not have had a chance to create a master record for this customer.

Use the statements written in Exercise 11.3 as the basis for a complete file-matching accounts receivable program. Use the account number on each file as the record key for matching purposes. Assume that each file is a sequential file with records stored in increasing account number order.

When a match occurs (i.e., records with the same account number appear on both the master file and the transaction file), add the dollar amount on the transaction file to the current balance on the master file and write the "newmast.dat" record. (Assume that purchases are indicated by positive amounts on the transaction file, and that payments are indicated by negative amounts.)
When there is a master record for a particular account but no corresponding transaction record, merely write the master record to "newmast.dat". When there is a transaction record but no corresponding master record, print the message "Unmatched transaction record for account number ..." (fill in the account number from the transaction record).

11.8 (Creating Data for the File Matching Program) After writing the program of Exercise 11.7, write a simple program to create some test data for checking out the program of Exercise 11.7. Use the following sample account data:

### Master File:

<table>
<thead>
<tr>
<th>Account number</th>
<th>Name</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Alan Jones</td>
<td>348.17</td>
</tr>
<tr>
<td>300</td>
<td>Mary Smith</td>
<td>27.19</td>
</tr>
<tr>
<td>500</td>
<td>Sam Sharp</td>
<td>0.00</td>
</tr>
<tr>
<td>700</td>
<td>Suzy Green</td>
<td>-14.22</td>
</tr>
</tbody>
</table>

### Transaction File:

<table>
<thead>
<tr>
<th>Account number</th>
<th>Dollar amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>27.14</td>
</tr>
<tr>
<td>300</td>
<td>62.11</td>
</tr>
<tr>
<td>400</td>
<td>100.56</td>
</tr>
<tr>
<td>900</td>
<td>82.17</td>
</tr>
</tbody>
</table>

11.9 Run the program of Exercise 11.7 using the files of test data created in Exercise 11.8. Use the listing program of Exercise 11.7 to print the new master file. Check the results carefully.

11.10 (File Matching with Multiple Transactions) It’s possible (actually common) to have several transaction records with the same record key. This occurs because a particular customer might make several purchases and cash payments during a business period. Rewrite your accounts receivable file-matching program of Exercise 11.7 to provide for the possibility of handling several transaction records with the same record key. Modify the test data of Exercise 11.8 to include the following additional transaction records:

### Additional Transaction Records:

<table>
<thead>
<tr>
<th>Account number</th>
<th>Dollar amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>83.89</td>
</tr>
<tr>
<td>700</td>
<td>80.78</td>
</tr>
<tr>
<td>700</td>
<td>1.53</td>
</tr>
</tbody>
</table>

11.11 Write statements that accomplish each of the following. Assume that the structure

```c
struct person {
  char lastName[ 15 ];
  char firstName[ 15 ];
  char age[ 4 ];
};
```

has been defined and that the file is already open for writing.
Exercises

a) Initialize the file "nameage.dat" so that there are 100 records with lastName = "unassigned", firstname = "" and age = "0".
b) Input 10 last names, first names and ages, and write them to the file.
c) Update a record; if there is no information in the record, tell the user "No info".
d) Delete a record that has information by reinitializing that particular record.

11.12 (Hardware Inventory) You’re the owner of a hardware store and need to keep an inventory that can tell you what tools you have, how many you have and the cost of each one. Write a program that initializes the file "hardware.dat" to 100 empty records, lets you input the data concerning each tool, enables you to list all your tools, lets you delete a record for a tool that you no longer have and lets you update any information in the file. The tool identification number should be the record number. Use the following information to start your file:

<table>
<thead>
<tr>
<th>Record #</th>
<th>Tool name</th>
<th>Quantity</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Electric sander</td>
<td>7</td>
<td>57.98</td>
</tr>
<tr>
<td>17</td>
<td>Hammer</td>
<td>76</td>
<td>11.99</td>
</tr>
<tr>
<td>24</td>
<td>Jig saw</td>
<td>21</td>
<td>11.00</td>
</tr>
<tr>
<td>39</td>
<td>Lawn mower</td>
<td>3</td>
<td>79.50</td>
</tr>
<tr>
<td>56</td>
<td>Power saw</td>
<td>18</td>
<td>99.99</td>
</tr>
<tr>
<td>68</td>
<td>Screwdriver</td>
<td>106</td>
<td>6.99</td>
</tr>
<tr>
<td>77</td>
<td>Sledge hammer</td>
<td>11</td>
<td>21.50</td>
</tr>
<tr>
<td>83</td>
<td>Wrench</td>
<td>34</td>
<td>7.50</td>
</tr>
</tbody>
</table>

11.13 (Telephone Number Word Generator) Standard telephone keypads contain the digits 0 through 9. The numbers 2 through 9 each have three letters associated with them, as is indicated by the following table:

<table>
<thead>
<tr>
<th>Digit</th>
<th>Letter</th>
<th>Digit</th>
<th>Letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>A</td>
<td>6</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>7</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>8</td>
<td>O</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>9</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td></td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>G</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td></td>
<td>U</td>
</tr>
<tr>
<td></td>
<td>I</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>J</td>
<td></td>
<td>W</td>
</tr>
<tr>
<td></td>
<td>K</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>

Many people find it difficult to memorize phone numbers, so they use the correspondence between digits and letters to develop seven-letter words that correspond to their phone numbers. For example, a person whose telephone number is 686-2377 might use the correspondence indicated in the above table to develop the seven-letter word “NUMBERS.”

Businesses frequently attempt to get telephone numbers that are easy for their clients to remember. If a business can advertise a simple word for its customers to dial, then, no doubt, the business will receive a few more calls.

Each seven-letter word corresponds to exactly one seven-digit telephone number. The restaurant wishing to increase its take-home business could surely do so with the number 825-3688 (i.e., “TAKEOUT”).

Each seven-digit phone number corresponds to many separate seven-letter words. Unfortunately, most of these represent unrecognizable juxtapositions of letters. It’s possible, however, that the owner of a barber shop would be pleased to know that the shop’s telephone number, 424-7288, corresponds to “HAIRCUT.” The owner of a liquor store would, no doubt, be delighted to find
Chapter 11  C File Processing

Write a C program that, given a seven-digit number, writes to a file every possible seven-letter word corresponding to that number. There are 2187 (3 to the seventh power) such words. Avoid phone numbers with the digits 0 and 1.

11.14  *(Telephone Number Word Generator Modification)* If you have a computerized dictionary available, modify the program you wrote in Exercise 11.13 to look up the words in the dictionary. Some seven-letter combinations created by this program consist of two or more words (the phone number 843-2677 produces “THEBOSS”).

11.15  Modify the example of Fig. 8.14 to use functions `fgetc` and `fputs` rather than `getchar` and `puts`. The program should give the user the option to read from the standard input and write to the standard output or to read from a specified file and write to a specified file. If the user chooses the second option, have the user enter the file names for the input and output files.

11.16  *(Outputting Type Sizes to a File)* Write a program that uses the `sizeof` operator to determine the sizes in bytes of the various data types on your computer system. Write the results to the file “datasize.dat” so you may print the results later. The format for the results in the file should be as follows:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>16</td>
</tr>
</tbody>
</table>

[Note: The type sizes on your computer might be different from those listed above.]

11.17  *(Simpletron with File Processing)* In Exercise 7.19, you wrote a software simulation of a computer that used a special machine language called Simpletron Machine Language (SML). In the simulation, each time you wanted to run an SML program, you entered the program into the simulator from the keyboard. If you made a mistake while typing the SML program, the simulator was restarted and the SML code was reentered. It would be nice to be able to read the SML program from a file rather than type it each time. This would reduce time and mistakes in preparing to run SML programs.

   a) Modify the simulator you wrote in Exercise 7.19 to read SML programs from a file specified by the user at the keyboard.

   b) After the Simpletron executes, it outputs the contents of its registers and memory on the screen. It would be nice to capture the output in a file, so modify the simulator to write its output to a file in addition to displaying the output on the screen.

Making a Difference

11.1  *(Phishing Scanner)* Phishing is a form of identity theft in which, in an e-mail, a sender posing as a trustworthy source attempts to acquire private information, such as your user names, passwords, credit-card numbers and social security number. Phishing e-mails claiming to be from
popular banks, credit-card companies, auction sites, social networks and online payment services may look quite legitimate. These fraudulent messages often provide links to spoofed (fake) websites where you’re asked to enter sensitive information.


Create a list of 30 words, phrases and company names commonly found in phishing messages. Assign a point value to each based on your estimate of its likeliness to be in a phishing message (e.g., one point if it’s somewhat likely, two points if moderately likely, or three points if highly likely). Write a program that scans a file of text for these terms and phrases. For each occurrence of a keyword or phrase within the text file, add the assigned point value to the total points for that word or phrase. For each keyword or phrase found, output one line with the word or phrase, the number of occurrences and the point total. Then show the point total for the entire message. Does your program assign a high point total to some actual phishing e-mails you’ve received? Does it assign a high point total to some legitimate e-mails you’ve received?
Much that I bound,
I could not free;
Much that I freed
returned to me.
—Lee Wilson Dodd

‘Will you walk a little faster?’
said a whiting to a snail, ‘There’s
a porpoise close behind us, and
he’s treading on my tail.’
—Lewis Carroll

There is always room at the top.
—Daniel Webster

Push on — keep moving.
—Thomas Morton

I think that I shall never see
A poem lovely as a tree.
—Joyce Kilmer

Objectives
In this chapter, you’ll learn:

■ To allocate and free memory
dynamically for data objects.

■ To form linked data structures
using pointers, self-referential
structures and recursion.

■ To create and manipulate
linked lists, queues, stacks
and binary trees.

■ Various important
applications of linked data
structures.
12.1 Introduction

We’ve studied fixed-size data structures such as single-subscripted arrays, double-subscripted arrays and structs. This chapter introduces dynamic data structures with sizes that grow and shrink at execution time. Linked lists are collections of data items “lined up in a row”—insertions and deletions are made anywhere in a linked list. Stacks are important in compilers and operating systems—insertions and deletions are made only at one end of a stack—its top. Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and deletions are made from the front (also referred to as the head) of a queue. Binary trees facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representing file system directories and compiling expressions into machine language. Each of these data structures has many other interesting applications.

We’ll discuss each of the major types of data structures and implement programs that create and manipulate these data structures. In the next part of the book—the introduction to C++ and object-oriented programming—we’ll study data abstraction. This technique will enable us to build these data structures in a dramatically different manner designed for producing software that is much easier to maintain and reuse.

This is a challenging chapter. The programs are substantial and they incorporate most of what you have learned in the earlier chapters. The programs are especially heavy on pointer manipulation, a subject many people consider to be among the most difficult topics in C. The chapter is loaded with highly practical programs that you’ll be able to use in more advanced courses; the chapter includes a rich collection of exercises that emphasize practical applications of the data structures.

We sincerely hope that you’ll attempt the major project described in the special section entitled Building Your Own Compiler. You have been using a compiler to translate your C programs to machine language so that you could execute your programs on your computer. In this project, you’ll actually build your own compiler. It will read a file of statements written in a simple, yet powerful, high-level language similar to early versions of the popular language BASIC. Your compiler will translate these statements into a file of Simpletron Machine Language (SML) instructions. Your Simpletron Simulator program will then execute the SML program produced by your compiler! This project will give you a wonderful opportunity to exercise most of what you have learned in this course. The special section carefully walks you through the specifications of the high-level language, and describes the algorithms you’ll need to convert each type of high-level language statement into machine language instructions. If you enjoy being challenged, you might
attempt the many enhancements to both the compiler and the Simpletron Simulator suggested in the exercises.

### 12.2 Self-Referential Structures

A self-referential structure contains a pointer member that points to a structure of the same structure type. For example, the definition

```c
struct node {
    int data;
    struct node *nextPtr;
};
```

defines a type, struct node. A structure of type struct node has two members—integer member data and pointer member nextPtr. Member nextPtr points to a structure of type struct node—a structure of the same type as the one being declared here, hence the term “self-referential structure.” Member nextPtr is referred to as a link—i.e., nextPtr can be used to “tie” a structure of type struct node to another structure of the same type. Self-referential structures can be linked together to form useful data structures such as lists, queues, stacks and trees. Figure 12.1 illustrates two self-referential structure objects linked together to form a list. A slash—representing a NULL pointer—is placed in the link member of the second self-referential structure to indicate that the link does not point to another structure. [Note: The slash is only for illustration purposes; it does not correspond to the backslash character in C.] A NULL pointer normally indicates the end of a data structure just as the null character indicates the end of a string.

#### Common Programming Error 12.1

Not setting the link in the last node of a list to NULL can lead to runtime errors.

---

### 12.3 Dynamic Memory Allocation

Creating and maintaining dynamic data structures requires dynamic memory allocation—the ability for a program to obtain more memory space at execution time to hold new nodes, and to release space no longer needed. The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available virtual memory in a virtual memory system. Often, the limits are much smaller because available memory must be shared among many applications.

Functions malloc and free, and operator sizeof, are essential to dynamic memory allocation. Function malloc takes as an argument the number of bytes to be allocated and returns a pointer of type void * (pointer to void) to the allocated memory. A void * pointer may be assigned to a variable of any pointer type. Function malloc is normally used with the sizeof operator. For example, the statement
newPtr = malloc( sizeof( struct node ) );

evaluates `sizeof(struct node)` to determine the size in bytes of a structure of type `struct node`, allocates a new area in memory of that number of bytes and stores a pointer to the allocated memory in variable `newPtr`. The allocated memory is not initialized. If no memory is available, `malloc` returns `NULL`.

Function `free` deallocates memory—i.e., the memory is returned to the system so that the memory can be reallocated in the future. To free memory dynamically allocated by the preceding `malloc` call, use the statement

```
free( newPtr );
```

C also provides functions `calloc` and `realloc` for creating and modifying dynamic arrays. These functions are discussed in Section 14.11. The following sections discuss lists, stacks, queues and trees, each of which is created and maintained with dynamic memory allocation and self-referential structures.

**Portability Tip 12.1**

A structure’s size is not necessarily the sum of the sizes of its members. This is so because of various machine-dependent boundary alignment requirements (see Chapter 10).

**Common Programming Error 12.2**

Assuming that the size of a structure is simply the sum of the sizes of its members is a logic error.

**Good Programming Practice 12.1**

Use the `sizeof` operator to determine the size of a structure.

**Error-Prevention Tip 12.1**

When using `malloc`, test for a NULL pointer return value, which indicates that the memory was not allocated.

**Common Programming Error 12.3**

Not returning dynamically allocated memory when it’s no longer needed can cause the system to run out of memory prematurely. This is sometimes called a “memory leak.”

**Good Programming Practice 12.2**

When memory that was dynamically allocated is no longer needed, use `free` to return the memory to the system immediately.

**Common Programming Error 12.4**

Freeing memory not allocated dynamically with `malloc` is an error.

**Common Programming Error 12.5**

Referring to memory that has been freed is an error that typically results in the program crashing.
12.4 Linked Lists

A linked list is a linear collection of self-referential structures, called nodes, connected by pointer links—hence, the term “linked” list. A linked list is accessed via a pointer to the first node of the list. Subsequent nodes are accessed via the link pointer member stored in each node. By convention, the link pointer in the last node of a list is set to NULL to mark the end of the list. Data is stored in a linked list dynamically—each node is created as necessary. A node can contain data of any type including other struct objects. Stacks and queues are also linear data structures, and, as we’ll see, are constrained versions of linked lists. Trees are nonlinear data structures.

Lists of data can be stored in arrays, but linked lists provide several advantages. A linked list is appropriate when the number of data elements to be represented in the data structure is unpredictable. Linked lists are dynamic, so the length of a list can increase or decrease as necessary. The size of an array, however cannot be altered once memory is allocated. Arrays can become full. Linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

Performance Tip 12.1
An array can be declared to contain more elements than the number of data items expected, but this can waste memory. Linked lists can provide better memory utilization in these situations.

Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list.

Performance Tip 12.2
Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately.

Performance Tip 12.3
The elements of an array are stored contiguously in memory. This allows immediate access to any array element because the address of any element can be calculated directly based on its position relative to the beginning of the array. Linked lists do not afford such immediate access to their elements.

Linked list nodes are normally not stored contiguously in memory. Logically, however, the nodes of a linked list appear to be contiguous. Figure 12.2 illustrates a linked list with several nodes.

![Fig. 12.2](image)

Linked list graphical representation.
Performance Tip 12.4

Using dynamic memory allocation (instead of arrays) for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that the pointers take up space, and that dynamic memory allocation incurs the overhead of function calls.

Figure 12.3 (output shown in Fig. 12.4) manipulates a list of characters. The program enables you to insert a character in the list in alphabetical order (function insert) or to delete a character from the list (function delete). This is a large and complex program. A detailed discussion of the program follows. Exercise 12.20 asks the reader to implement a recursive function that prints a list backwards. Exercise 12.21 asks the reader to implement a recursive function that searches a linked list for a particular data item.

```c
/* Fig. 12.3: fig12_03.c
   Operating and maintaining a list */
#include <stdio.h>
#include <stdlib.h>

/* self-referential structure */
struct ListNode {
    char data; /* each ListNode contains a character */
    struct ListNode *nextPtr; /* pointer to next node */
}; /* end structure ListNode */

typedef struct ListNode ListNode; /* synonym for struct ListNode */
typedef ListNode *ListNodePtr; /* synonym for ListNode* */

/* prototypes */
void insert( ListNodePtr *sPtr, char value );
char delete( ListNodePtr *sPtr, char value );
int isEmpty( ListNodePtr sPtr );
void printList( ListNodePtr currentPtr );
void instructions( void );

int main( void )
{
    ListNodePtr startPtr = NULL; /* initially there are no nodes */
    int choice; /* user's choice */
    char item; /* char entered by user */

    instructions(); /* display the menu */
    printf("? ");
    scanf("%d", &choice);

    /* loop while user does not choose 3 */
    while ( choice != 3 ) {
        switch ( choice ) {
        case 1:
            printf("Enter a character: ");
            scanf("\n%c", &item);
```
```c
insert( &startPtr, item ); /* insert item in list */
printList( startPtr );
br
break;

case 2: /* delete an element */
/* if list is not empty */
if ( !isEmpty( startPtr ) ) {
    printf( "Enter character to be deleted: " );
    scanf( "\n%c", &item );

    /* if character is found, remove it*/
    if ( delete( &startPtr, item ) ) { /* remove item */
        printf( "%c deleted.\n", item );
        printList( startPtr );
    } /* end if */
    else {
        printf( "%c not found.\n\n", item );
    } /* end else */
} /* end if */
else {
    printf( "List is empty.\n\n" );
} /* end else */
break;
default:
    printf( "Invalid choice.\n\n" );
    instructions();
br
} /* end switch */

printf( "? " );
scanf( "%d", &choice );
} /* end while */

printf( "End of run.\n" );
return 0; /* indicates successful termination */
} /* end main */

/* display program instructions to user */
void instructions( void )
{
    printf( "Enter your choice:\n"
        " 1 to insert an element into the list.\n"
        " 2 to delete an element from the list.\n"
        " 3 to end.\n" );
} /* end function instructions */

/* Insert a new value into the list in sorted order */
void insert( ListNodePtr *sPtr, char value )
{
    ListNodePtr newPtr; /* pointer to new node */
    ListNodePtr previousPtr; /* pointer to previous node in list */
    ListNodePtr currentPtr; /* pointer to current node in list */
```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 2 of 4.)
newPtr = malloc( sizeof( ListNode ) ); /* create node */
if ( newPtr != NULL ) { /* is space available */
    newPtr->data = value; /* place value in node */
    newPtr->nextPtr = NULL; /* node does not link to another node */
    previousPtr = NULL;
    currentPtr = *sPtr;
    /* loop to find the correct location in the list */
    while ( currentPtr != NULL && value > currentPtr->data ) {
        previousPtr = currentPtr; /* walk to ... */
        currentPtr = currentPtr->nextPtr; /* ... next node */
    } /* end while */
    /* insert new node at beginning of list */
    if ( previousPtr == NULL ) {
        newPtr->nextPtr = *sPtr;
        *sPtr = newPtr;
    } /* end if */
    else {
        /* insert new node between previousPtr and currentPtr */
        previousPtr->nextPtr = newPtr;
        newPtr->nextPtr = currentPtr;
    } /* end else */
} /* end if */
else {
    printf("%c not inserted. No memory available.\n", value);
} /* end else */
} /* end function insert */

/* Delete a list element */
char delete( ListNodePtr *sPtr, char value )
{
    ListNodePtr previousPtr; /* pointer to previous node in list */
    ListNodePtr currentPtr; /* pointer to current node in list */
    ListNodePtr tempPtr; /* temporary node pointer */
    /* delete first node */
    if ( value == ( *sPtr )->data ) {
        tempPtr = *sPtr; /* hold onto node being removed */
        *sPtr = ( *sPtr )->nextPtr; /* de-thread the node */
        free( tempPtr ); /* free the de-threaded node */
        return value;
    } /* end if */
    else {
        previousPtr = *sPtr;
        currentPtr = ( *sPtr )->nextPtr;
        /* loop to find the correct location in the list */
        while ( currentPtr != NULL && currentPtr->data != value ) {
            previousPtr = currentPtr; /* walk to ... */
            currentPtr = currentPtr->nextPtr; /* ... next node */
        } /* end while */
    } /* end else */
/* delete node at currentPtr */
if ( currentPtr != NULL ) {
    tempPtr = currentPtr;
    previousPtr->nextPtr = currentPtr->nextPtr;
    free( tempPtr );
    return value;
} /* end if */
} /* end else */
return '\0';
} /* end function delete */

/* Return 1 if the list is empty, 0 otherwise */
int isEmpty( ListNodePtr sPtr )
{
    return sPtr == NULL;
} /* end function isEmpty */

/* Print the list */
void printList( ListNodePtr currentPtr )
{
    /* if list is empty */
    if ( currentPtr == NULL ) {
        printf( "List is empty.\n\n" );
    } /* end if */
    else {
        printf( "The list is:\n" );
        /* while not the end of the list */
        while ( currentPtr != NULL ) {
            printf( "%c --> ", currentPtr->data );
            currentPtr = currentPtr->nextPtr;
        } /* end while */
        printf( "NULL\n\n" );
    } /* end else */
} /* end function printList */

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 4 of 4.)

Enter your choice:
1 to insert an element into the list.
2 to delete an element from the list.
3 to end.
? 1
Enter a character: B
The list is:
B --> NULL

? 1
Enter a character: A

Fig. 12.4 | Sample output for the program of Fig. 12.3. (Part 1 of 2.)
The primary functions of linked lists are insert (lines 86–120) and delete (lines 123–156). Function isEmpty (lines 159–162) is called a predicate function—it does not alter the list in any way; rather it determines if the list is empty (i.e., the pointer to the first node of the list is NULL). If the list is empty, 1 is returned; otherwise, 0 is returned. Function printList (lines 165–182) prints the list.

**Function insert**

Characters are inserted in the list in alphabetical order. Function insert (lines 86–120) receives the address of the list and a character to be inserted. The address of the list is necessary when a value is to be inserted at the start of the list. Providing the address of the list enables the list (i.e., the pointer to the first node of the list) to be modified via a call by reference. Since the list itself is a pointer (to its first element), passing the address of the list creates a pointer to a pointer (i.e., double indirection). This is a complex notion and
requires careful programming. The steps for inserting a character in the list are as follows (see Fig. 12.5):

1. Create a node by calling malloc, assigning to newPtr the address of the allocated memory (line 92), assigning the character to be inserted to newPtr->data (line 95), and assigning NULL to newPtr->nextPtr (line 96).

2. Initialize previousPtr to NULL (line 198) and currentPtr to *sPtr (line 99)—the pointer to the start of the list. Pointers previousPtr and currentPtr store the locations of the node preceding the insertion point and the node after the insertion point.

3. While currentPtr is not NULL and the value to be inserted is greater than currentPtr->data (line 102), assign currentPtr to previousPtr (line 103) and advance currentPtr to the next node in the list (line 104). This locates the insertion point for the value.

4. If previousPtr is NULL (line 108), insert the new node as the first node in the list (lines 109–110). Assign *sPtr to newPtr->nextPtr (the new node link points to the former first node) and assign newPtr to *sPtr (*sPtr points to the new node). Otherwise, if previousPtr is not NULL, the new node is inserted in place (lines 113–114). Assign newPtr to previousPtr->nextPtr (the previous node points to the new node) and assign currentPtr to newPtr->nextPtr (the new node link points to the current node).

---

**Fig. 12.5** | Inserting a node in order in a list.
Figure 12.5 illustrates the insertion of a node containing the character 'C' into an ordered list. Part a) of the figure shows the list and the new node before the insertion. Part b) of the figure shows the result of inserting the new node. The reassigned pointers are dotted arrows. For simplicity, we implemented function `insert` (and other similar functions in this chapter) with a `void` return type. It’s possible that function `malloc` will fail to allocate the requested memory. In this case, it would be better for our `insert` function to return a status that indicates whether the operation was successful.

**Function delete**

Function `delete` (lines 123–156) receives the address of the pointer to the start of the list and a character to be deleted. The steps for deleting a character from the list are as follows:

1. If the character to be deleted matches the character in the first node of the list (line 130), assign `*sPtr` to `tempPtr` (`tempPtr` will be used to free the unneeded memory), assign `(*sPtr)->nextPtr` to `*sPtr` (`*sPtr` now points to the second node in the list), free the memory pointed to by `tempPtr`, and return the character that was deleted.

2. Otherwise, initialize `previousPtr` with `*sPtr` and initialize `currentPtr` with `(*sPtr)->nextPtr` (lines 137–138).

3. While `currentPtr` is not `NULL` and the value to be deleted is not equal to `currentPtr->data` (Line 141), assign `currentPtr` to `previousPtr` (line 142), and assign `currentPtr->nextPtr` to `currentPtr` (line 143). This locates the character to be deleted if it’s contained in the list.

4. If `currentPtr` is not `NULL` (line 147), assign `currentPtr` to `tempPtr` (line 148), assign `currentPtr->nextPtr` to `previousPtr->nextPtr` (line 149), free the node pointed to by `tempPtr` (line 150), and return the character that was deleted from the list (line 151). If `currentPtr` is `NULL`, return the null character (`'\0'`) to signify that the character to be deleted was not found in the list (line 155).

Figure 12.6 illustrates the deletion of a node from a linked list. Part a) of the figure shows the linked list after the preceding insert operation. Part b) shows the reassignment of the link element of `previousPtr` and the assignment of `currentPtr` to `tempPtr`. Pointer `tempPtr` is used to free the memory allocated to store 'C'.

**Function printList**

Function `printList` (lines 165–182) receives a pointer to the start of the list as an argument and refers to the pointer as `currentPtr`. The function first determines if the list is empty (lines 168–170) and, if so, prints "The list is empty." and terminates. Otherwise, it prints the data in the list (lines 171–181). While `currentPtr` is not `NULL`, the value of `currentPtr->data` is printed by the function, and `currentPtr->nextPtr` is assigned to `currentPtr`. If the link in the last node of the list is not `NULL`, the printing algorithm will try to print past the end of the list, and an error will occur. The printing algorithm is identical for linked lists, stacks and queues.

---

**Error-Prevention Tip 12.2**

Assign `NULL` to the link member of a new node. Pointers should be initialized before they are used.
Chapter 12  C Data Structures

12.5 Stacks

A stack is a constrained version of a linked list. New nodes can be added to a stack and removed from a stack only at the top. For this reason, a stack is referred to as a last-in, first-out (LIFO) data structure. A stack is referenced via a pointer to the top element of the stack. The link member in the last node of the stack is set to NULL to indicate the bottom of the stack.

Figure 12.7 illustrates a stack with several nodes. Stacks and linked lists are represented identically. The difference between stacks and linked lists is that insertions and deletions may occur anywhere in a linked list, but only at the top of a stack.

Common Programming Error 12.6

Not setting the link in the bottom node of a stack to NULL can lead to runtime errors.

Fig. 12.7  Stack graphical representation.

The primary functions used to manipulate a stack are push and pop. Function push creates a new node and places it on top of the stack. Function pop removes a node from the top of the stack, frees the memory that was allocated to the popped node and returns the popped value.
Figure 12.8 (output shown in Fig. 12.9) implements a simple stack of integers. The program provides three options: 1) push a value onto the stack (function `push`), 2) pop a value off the stack (function `pop`) and 3) terminate the program.

```c
/* Fig. 12.8: fig12_08.c */
#include <stdio.h>
#include <stdlib.h>

/* self-referential structure */
struct stackNode {
    int data; /* define data as an int */
    struct stackNode *nextPtr; /* stackNode pointer */
}; /* end structure stackNode */

typedef struct stackNode StackNode; /* synonym for struct stackNode */
typedef StackNode *StackNodePtr; /* synonym for StackNode* */

/* prototypes */
void push( StackNodePtr *topPtr, int info );
int pop( StackNodePtr *topPtr );
int isEmpty( StackNodePtr topPtr );
void printStack( StackNodePtr currentPtr );
void instructions( void );

/* function main begins program execution */
int main( void )
{
    StackNodePtr stackPtr = NULL; /* points to stack top */
    int choice; /* user's menu choice */
    int value; /* int input by user */

    instructions(); /* display the menu */
    printf( "? ");
    scanf( "%d", &choice );

    /* while user does not enter 3 */
    while ( choice != 3 ){

        switch ( choice ) {
        /* push value onto stack */
        case 1:
            printf( "Enter an integer: " );
            scanf( "%d", &value );
            push( &stackPtr, value );
            printStack( stackPtr );
            break;
        /* pop value off stack */
        case 2:
            /* if stack is not empty */
            if ( !isEmpty( stackPtr ) ) {
```
Chapter 12  C Data Structures

Fig. 12.8  |  A simple stack program. (Part 2 of 3.)

printf( "The popped value is %d.\n", pop( &stackPtr ) );

printStack( stackPtr );
break;
default:
printf( "Invalid choice.\n\n" );
instructions();
break;
} /* end switch */

printf( "? ");
scanf( "%d", &choice );
} /* end while */

printf( "End of run.\n" );
return 0; /* indicates successful termination */
} /* end main */

/* display program instructions to user */
void instructions( void )
{
    printf( "Enter choice:\n"
        "1 to push a value on the stack\n"
        "2 to pop a value off the stack\n"
        "3 to end program\n" );
} /* end function instructions */

/* Insert a node at the stack top */
void push( StackNodePtr *topPtr, int info ) 
{
    StackNodePtr newPtr; /* pointer to new node */
    
    newPtr = malloc( sizeof( StackNode ) );

    /* insert the node at stack top */
    if ( newPtr != NULL ) {
        newPtr->data = info;
        newPtr->nextPtr = *topPtr;
        *topPtr = newPtr;
    } /* end if */
    else { /* no space available */
        printf( "%d not inserted. No memory available.\n", info );
    } /* end else */
} /* end function push */

/* Remove a node from the stack top */
int pop( StackNodePtr *topPtr )
{
    StackNodePtr tempPtr; /* temporary node pointer */
    int popValue; /* node value */

    newPtr = malloc( sizeof( StackNode ) );
tempPtr = *topPtr;
popValue = ( *topPtr )->data;
topPtr = ( *topPtr )->nextPtr;
free( tempPtr );
return popValue;
} /* end function pop */

/* Print the stack */

void printStack( StackNodePtr currentPtr )
{
/* if stack is empty */
if ( currentPtr == NULL ) {
printf( "The stack is empty.\n\n" );
} /* end if */
else {
printf( "The stack is:\n" );
/* while not the end of the stack */
while ( currentPtr != NULL ) {
printf( "%d --> ", currentPtr->data );
currentPtr = currentPtr->nextPtr;
} /* end while */
printf( "NULL\n\n" );
} /* end else */
/* end function printList */

/* Return 1 if the stack is empty, 0 otherwise */
int isEmpty( StackNodePtr topPtr )
{
return topPtr == NULL;
} /* end function isEmpty */

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL

? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

Fig. 12.8 | A simple stack program. (Part 3 of 3.)

Fig. 12.9 | Sample output from the program of Fig. 12.8. (Part 1 of 2.)
Function push
Function push (lines 77–92) places a new node at the top of the stack. The function consists of three steps:

1. Create a new node by calling malloc and assign the location of the allocated memory to newPtr (line 81).
2. Assign to newPtr->data the value to be placed on the stack (line 85) and assign *topPtr (the stack top pointer) to newPtr->nextPtr (line 86)—the link member of newPtr now points to the previous top node.
3. Assign newPtr to *topPtr (line 87)—*topPtr now points to the new stack top.

Manipulations involving *topPtr change the value of stackPtr in main. Figure 12.10 illustrates function push. Part a) of the figure shows the stack and the new node before the push operation. The dotted arrows in part b) illustrate Steps 2 and 3 of the push operation that enable the node containing 12 to become the new stack top.

Function pop
Function pop (lines 95–105) removes a node from the top of the stack. Function main determines if the stack is empty before calling pop. The pop operation consists of five steps:

1. Assign *topPtr to tempPtr (line 100); tempPtr will be used to free the unneeded memory.
2. Assign (*topPtr)->data to popValue (line 101) to save the value in the top node.
3. Assign (*topPtr)->nextPtr to *topPtr (line 102) so *topPtr contains address of the new top node.

4. Free the memory pointed to by tempPtr (line 103).

5. Return popValue to the caller (line 104).

Figure 12.11 illustrates function pop. Part (a) shows the stack after the previous push operation. Part (b) shows tempPtr pointing to the first node of the stack and topPtr pointing to the second node of the stack. Function free is used to free the memory pointed to by tempPtr.

Applications of Stacks
Stacks have many interesting applications. For example, whenever a function call is made, the called function must know how to return to its caller, so the return address is pushed
onto a stack. If a series of function calls occurs, the successive return values are pushed onto
the stack in last-in, first-out order so that each function can return to its caller. Stacks sup-
port recursive function calls in the same manner as conventional nonrecursive calls.

Stacks contain the space created for automatic variables on each invocation of a func-
tion. When the function returns to its caller, the space for that function's automatic vari-
ables is popped off the stack, and these variables no longer are known to the program.
Stacks are used by compilers in the process of evaluating expressions and generating
machine language code. The exercises explore several applications of stacks.

12.6 Queues

Another common data structure is the queue. A queue is similar to a checkout line in a
grocery store—the first person in line is serviced first, and other customers enter the line
only at the end and wait to be serviced. Queue nodes are removed only from the head of the queue and are inserted only at the tail of the queue. For this reason, a queue is referred
to as a first-in, first-out (FIFO) data structure. The insert and remove operations are
known as enqueue and dequeue.

Queues have many applications in computer systems. Many computers have only a
single processor, so only one user at a time may be serviced. Entries for the other users are
placed in a queue. Each entry gradually advances to the front of the queue as users receive
service. The entry at the front of the queue is the next to receive service.

Queues are also used to support print spooling. A multiuser environment may have
only a single printer. Many users may be generating outputs to be printed. If the printer
is busy, other outputs may still be generated. These are spooled to disk where they wait in
a queue until the printer becomes available.

Information packets also wait in queues in computer networks. Each time a packet
arrives at a network node, it must be routed to the next node on the network along the
path to the packet's final destination. The routing node routes one packet at a time, so
additional packets are enqueued until the router can route them. Figure 12.12 illustrates
a queue with several nodes. Note the pointers to the head of the queue and the tail of the
queue.

**Common Programming Error 12.7**

Not setting the link in the last node of a queue to NULL can lead to runtime errors.

![Fig. 12.12](image-url) | Queue graphical representation.
Figure 12.13 (output in Fig. 12.14) performs queue manipulations. The program provides several options: insert a node in the queue (function `enqueue`), remove a node from the queue (function `dequeue`) and terminate the program.

```c
/* Fig. 12.13: fig12_13.c */
#include <stdio.h>
#include <stdlib.h>

/* self-referential structure */
struct queueNode {
    char data; /* define data as a char */
    struct queueNode *nextPtr; /* queueNode pointer */
}; /* end structure queueNode */

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

/* function prototypes */
void printQueue( QueueNodePtr currentPtr );
int isEmpty( QueueNodePtr headPtr );
char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr );
void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
              char value );
void instructions( void );

/* function main begins program execution */
int main( void )
{
    QueueNodePtr headPtr = NULL; /* initialize headPtr */
    QueueNodePtr tailPtr = NULL; /* initialize tailPtr */
    int choice; /* user's menu choice */
    char item; /* char input by user */

    instructions(); /* display the menu */
    printf( "? ");
    scanf( "%d", &choice );

    /* while user does not enter 3 */
    while ( choice != 3 )
    {
        switch( choice ) {
        /* enqueue value */
            case 1:
                printf( "Enter a character: " );
                scanf( "\n%c", &item );
                enqueue( &headPtr, &tailPtr, item );
                printQueue( headPtr );
                break;
        /* dequeue value */
```

Fig. 12.13 | Processing a queue. (Part 1 of 3.)
```c
case 2:
    /* if queue is not empty */
    if ( !isEmpty( headPtr ) ) {
        item = dequeue( &headPtr, &tailPtr);
        printf( "%c has been dequeued.\n", item );
    } /* end if */

    printQueue( headPtr );
    break;

default:
    printf( "Invalid choice.\n\n" );
    instructions();
    break;
} /* end switch */

printf( "? ");
scanf( "%d", &choice );
} /* end while */

printf( "End of run.\n" );
return 0; /* indicates successful termination */
} /* end main */

/* display program instructions to user */
void instructions( void ) {
    printf ( "Enter your choice:\n"
        "   1 to add an item to the queue\n"
        "   2 to remove an item from the queue\n"
        "   3 to end\n" );
} /* end function instructions */

/* insert a node a queue tail */
void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value ) {
    QueueNodePtr newPtr; /* pointer to new node */

    newPtr = malloc( sizeof( QueueNode ) );

    if ( newPtr != NULL ) { /* is space available */
        newPtr->data = value;
        newPtr->nextPtr = NULL;
    
        /* if empty, insert node at head */
        if ( isEmpty( *headPtr ) ) {
            *headPtr = newPtr;
        } /* end if */
        else {
            ( *tailPtr )->nextPtr = newPtr;
        } /* end else */
```
12.6 Queues

```c
*tailPtr = newPtr; /* end if */
else {
    printf("%c not inserted. No memory available.\n", value);
} /* end else */
} /* end function enqueue */

/* remove node from queue head */
char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
{
    char value; /* node value */
    QueueNodePtr tempPtr; /* temporary node pointer */

    value = ( *headPtr )->data;
    tempPtr = *headPtr;
    *headPtr = ( *headPtr )->nextPtr;

    /* if queue is empty */
    if ( *headPtr == NULL ) {
        *tailPtr = NULL;
    } /* end if */

    free( tempPtr );
    return value; /* end function dequeue */

/* Return 1 if the list is empty, 0 otherwise */
int isEmpty( QueueNodePtr headPtr )
{
    return headPtr == NULL; /* end function isEmpty */
}

/* Print the queue */
void printQueue( QueueNodePtr currentPtr )
{
    /* if queue is empty */
    if ( currentPtr == NULL ) {
        printf("Queue is empty.\n\n");
    } /* end if */
    else {
        printf("The queue is:\n");

        /* while not end of queue */
        while ( currentPtr != NULL ) {
            printf("%c --> ", currentPtr->data);
            currentPtr = currentPtr->nextPtr;
        } /* end while */

        printf("NULL\n\n");
    } /* end else */
} /* end function printQueue */
```

Fig. 12.13  |  Processing a queue. (Part 3 of 3.)
Enter your choice:
  1 to add an item to the queue
  2 to remove an item from the queue
  3 to end
?

Enter a character: A
The queue is:
A --> NULL

?

Enter a character: B
The queue is:
A --> B --> NULL

?

Enter a character: C
The queue is:
A --> B --> C --> NULL

?

Enter a character: A
A has been dequeued.
The queue is:
B --> C --> NULL

?

Enter a character: B
B has been dequeued.
The queue is:
C --> NULL

?

Enter a character: C
C has been dequeued.
Queue is empty.

?
Queue is empty.

?
Invalid choice.

Enter your choice:
  1 to add an item to the queue
  2 to remove an item from the queue
  3 to end
?

End of run.

**Fig. 12.14** | Sample output from the program in Fig. 12.13.

**Function enqueue**

Function enqueue (lines 80–104) receives three arguments from main: the address of the pointer to the head of the queue, the address of the pointer to the tail of the queue and the value to be inserted in the queue. The function consists of three steps:

1. To create a new node: Call malloc, assign the allocated memory location to `newPtr` (line 85), assign the value to be inserted in the queue to `newPtr->data` (line 88) and assign NULL to `newPtr->nextPtr` (line 89).
2. If the queue is empty (line 92), assign newPtr to *headPtr (line 93); otherwise, assign pointer newPtr to (*tailPtr)->nextPtr (line 96).

3. Assign newPtr to *tailPtr (line 99).

Figure 12.15 illustrates an enqueue operation. Part a) shows the queue and the new node before the operation. The dotted arrows in part b) illustrate Steps 2 and 3 of function enqueue that enable a new node to be added to the end of a queue that is not empty.

---

**Fig. 12.15** enqueue operation.

**Function dequeue**

Function dequeue (lines 107–123) receives the address of the pointer to the head of the queue and the address of the pointer to the tail of the queue as arguments and removes the first node from the queue. The dequeue operation consists of six steps:

1. Assign (*headPtr)->data to value to save the data (line 112).
2. Assign *headPtr to tempPtr (line 113), which will be used to free the unneeded memory.
3. Assign (*headPtr)->nextPtr to *headPtr (line 114) so that *headPtr now points to the new first node in the queue.
4. If *headPtr is NULL (line 117), assign NULL to *tailPtr (line 118).
5. Free the memory pointed to by tempPtr (line 121).
6. Return value to the caller (line 122).

Figure 12.16 illustrates function dequeue. Part a) shows the queue after the preceding enqueue operation. Part b) shows tempPtr pointing to the dequeued node, and headPtr pointing to the new first node of the queue. Function free is used to reclaim the memory pointed to by tempPtr.
12.7 Trees

Linked lists, stacks and queues are linear data structures. A tree is a nonlinear, two-dimensional data structure with special properties. Tree nodes contain two or more links. This section discusses binary trees (Fig. 12.17)—trees whose nodes all contain two links (none, one, or both of which may be NULL). The root node is the first node in a tree. Each link in the root node refers to a child. The left child is the first node in the left subtree, and the right child is the first node in the right subtree. The children of a node are called siblings. A node with no children is called a leaf node. Computer scientists normally draw trees from the root node down—exactly the opposite of trees in nature.
In this section, a special binary tree called a binary search tree is created. A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node, and the values in any right subtree are greater than the value in its parent node. Figure 12.18 illustrates a binary search tree with 12 values. The shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.

Common Programming Error 12.8
Not setting to NULL the links in leaf nodes of a tree can lead to runtime errors.

Figure 12.19 (output shown in Fig. 12.20) creates a binary search tree and traverses it three ways—inorder, preorder and postorder. The program generates 10 random numbers and inserts each in the tree, except that duplicate values are discarded.

```
/* Fig. 12.19: fig12_19.c
Create a binary tree and traverse it preorder, inorder, and postorder */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* prototypes */
void insertNode( TreeNodePtr *treePtr, int value );
void inOrder( TreeNodePtr treePtr );
void preOrder( TreeNodePtr treePtr );
void postOrder( TreeNodePtr treePtr );

/* self-referential structure */
struct treeNode {
    struct treeNode *leftPtr; /* pointer to left subtree */
    int data; /* node value */
    struct treeNode *rightPtr; /* pointer to right subtree */
}; /* end structure treeNode */

typedef struct treeNode TreeNode; /* synonym for struct treeNode */
typedef TreeNode *TreeNodePtr; /* synonym for TreeNode */

/* prototypes */
void insertNode( TreeNodePtr *treePtr, int value );
void inOrder( TreeNodePtr treePtr );
void preOrder( TreeNodePtr treePtr );
void postOrder( TreeNodePtr treePtr );
```

Fig. 12.18 | Binary search tree.

Fig. 12.19 | Creating and traversing a binary tree. (Part I of 3.)
/* function main begins program execution */
int main( void )
{
    int i; /* counter to loop from 1-10 */
    int item; /* variable to hold random values */
    TreeNodePtr rootPtr = NULL; /* tree initially empty */
    srand( time( NULL ) );
    printf( "The numbers being placed in the tree are:
" );
    /* insert random values between 0 and 14 in the tree */
    for ( i = 1; i <= 10; i++ ) {
        item = rand() % 15;
        printf( "%3d", item );
        insertNode( &rootPtr, item );
    } /* end for */
    /* traverse the tree preOrder */
    printf( "The preOrder traversal is:
" );
    preOrder( rootPtr );
    /* traverse the tree inOrder */
    printf( "The inOrder traversal is:
" );
    inOrder( rootPtr );
    /* traverse the tree postOrder */
    printf( "The postOrder traversal is:
" );
    postOrder( rootPtr );
    return 0; /* indicates successful termination */
} /* end main */

/* insert node into tree */
void insertNode( TreeNodePtr *treePtr, int value )
{
    /* if tree is empty */
    if ( *treePtr == NULL ) {
        *treePtr = malloc( sizeof( TreeNode ) );
        /* if memory was allocated then assign data */
        if ( *treePtr != NULL ) {
            ( *treePtr )->data = value;
            ( *treePtr )->leftPtr = NULL;
            ( *treePtr )->rightPtr = NULL;
        } /* end if */
    } /* end else */
    else {
        printf( "%d not inserted. No memory available.
", value );
    } /* end if */
} /* end if */
else {
    /* tree is not empty */
    /* data to insert is less than data in current node */
    if ( value < ( *treePtr )->data ) {
        insertNode( &( ( *treePtr )->leftPtr ), value );
    } /* end if */
/* data to insert is greater than data in current node */
else if ( value > ( *treePtr )->data ) {
    insertNode( &(( *treePtr )->rightPtr), value );
} /* end else if */
else { /* duplicate data value ignored */
    printf( "dup" );
} /* end else */
} /* end else */
} /* end function insertNode */

/* begin inorder traversal of tree */
void inOrder( TreeNodePtr treePtr )
{
    /* if tree is not empty then traverse */
    if ( treePtr != NULL ) {
        inOrder( treePtr->leftPtr );
        printf( "%3d", treePtr->data );
        inOrder( treePtr->rightPtr );
    } /* end if */
} /* end function inOrder */

/* begin preorder traversal of tree */
void preOrder( TreeNodePtr treePtr )
{
    /* if tree is not empty then traverse */
    if ( treePtr != NULL ) {
        printf( "%3d", treePtr->data );
        preOrder( treePtr->leftPtr );
        preOrder( treePtr->rightPtr );
    } /* end if */
} /* end function preOrder */

/* begin postorder traversal of tree */
void postOrder( TreeNodePtr treePtr )
{
    /* if tree is not empty then traverse */
    if ( treePtr != NULL ) {
        postOrder( treePtr->leftPtr );
        postOrder( treePtr->rightPtr );
        printf( "%3d", treePtr->data );
    } /* end if */
} /* end function postOrder */

Fig. 12.19  |  Creating and traversing a binary tree. (Part 3 of 3.)

The numbers being placed in the tree are:
6 7 4 12 7dup 2 2dup 5 7dup 11

The preOrder traversal is:
6 4 2 5 7 12 11

Fig. 12.20  |  Sample output from the program of Fig. 12.19. (Part 1 of 2.)
Chapter 12  C Data Structures

The functions used in Fig. 12.19 to create a binary search tree and traverse the tree are recursive. Function insertNode (lines 56–86) receives the address of the tree and an integer to be stored in the tree as arguments. A node can only be inserted as a leaf node in a binary search tree. The steps for inserting a node in a binary search tree are as follows:

1. If *treePtr is NULL (line 59), create a new node (line 60). Call malloc, assign the allocated memory to *treePtr, assign to (*treePtr)->data the integer to be stored (line 64), assign to (*treePtr)->leftPtr and (*treePtr)->rightPtr the value NULL (lines 65–66, and return control to the caller (either main or a previous call to insertNode).

2. If the value of *treePtr is not NULL and the value to be inserted is less than (*treePtr)->data, function insertNode is called with the address of (*treePtr)->leftPtr (line 75). If the value to be inserted is greater than (*treePtr)->data, function insertNode is called with the address of (*treePtr)->rightPtr (line 80). Otherwise, the recursive steps continue until a NULL pointer is found, then Step 1) is executed to insert the new node.

Functions inOrder (lines 89–97), preOrder (lines 100–108) and postOrder (lines 111–119) each receive a tree (i.e., the pointer to the root node of the tree) and traverse the tree. The steps for an inOrder traversal are:

1. Traverse the left subtree inOrder.
2. Process the value in the node.
3. Traverse the right subtree inOrder.

The value in a node is not processed until the values in its left subtree are processed. The inOrder traversal of the tree in Fig. 12.21 is:

6 13 17 27 33 42 48

The inOrder traversal of a binary search tree prints the node values in ascending order. The process of creating a binary search tree actually sorts the data—and thus this process is called the binary tree sort.

Fig. 12.20  Sample output from the program of Fig. 12.19. (Part 2 of 2.)

Fig. 12.21  Binary search tree with seven nodes.
The steps for a preorder traversal are:

1. Process the value in the node.
2. Traverse the left subtree preorder.
3. Traverse the right subtree preorder.

The value in each node is processed as the node is visited. After the value in a given node is processed, the values in the left subtree are processed, then the values in the right subtree are processed. The preorder traversal of the tree in Fig. 12.21 is:

```
27 13 6 17 42 33 48
```

The steps for a postorder traversal are:

1. Traverse the left subtree postorder.
2. Traverse the right subtree postorder.
3. Process the value in the node.

The value in each node is not printed until the values of its children are printed. The postorder traversal of the tree in Fig. 12.21 is:

```
6 17 13 33 48 42 27
```

The binary search tree facilitates duplicate elimination. As the tree is being created, an attempt to insert a duplicate value will be recognized because a duplicate will follow the same “go left” or “go right” decisions on each comparison as the original value did. Thus, the duplicate will eventually be compared with a node in the tree containing the same value. The duplicate value may simply be discarded at this point.

Searching a binary tree for a value that matches a key value is also fast. If the tree is tightly packed, each level contains about twice as many elements as the previous level. So a binary search tree with \( n \) elements would have a maximum of \( \log_2 n \) levels, and thus a maximum of \( \log_2 n \) comparisons would have to be made either to find a match or to determine that no match exists. This means, for example, that when searching a (tightly packed) 1000-element binary search tree, no more than 10 comparisons need to be made because \( 2^{10} > 1000 \). When searching a (tightly packed) 1,000,000 element binary search tree, no more than 20 comparisons need to be made because \( 2^{20} > 1,000,000 \).

In the exercises, algorithms are presented for several other binary tree operations such as deleting an item from a binary tree, printing a binary tree in a two-dimensional tree format and performing a level order traversal of a binary tree. The level order traversal of a binary tree visits the nodes of the tree row-by-row starting at the root node level. On each level of the tree, the nodes are visited from left to right. Other binary tree exercises include allowing a binary search tree to contain duplicate values, inserting string values in a binary tree and determining how many levels are contained in a binary tree.

**Summary**

*Section 12.1 Introduction*

- Dynamic data structures grow and shrink at execution time.
- Linked lists are collections of data items “lined up in a row”—insertions and deletions are made anywhere in a linked list.
• With stacks, insertions and deletions are made only at the top.
• Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and deletions are made from the front (also referred to as the head) of a queue.
• Binary trees facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representing file system directories and compiling expressions into machine language.

Section 12.2 Self-Referential Structures
• A self-referential structure contains a pointer member that points to a structure of the same type.
• Self-referential structures can be linked together to form lists, queues, stacks and trees.
• A **NULL** pointer normally indicates the end of a data structure.

Section 12.3 Dynamic Memory Allocation
• Creating and maintaining dynamic data structures require dynamic memory allocation.
• Functions **malloc** and **free**, and operator **sizeof**, are essential to dynamic memory allocation.
• Function **malloc** receives the number of bytes to be allocated and returns a **void *** pointer to the allocated memory. A **void *** pointer may be assigned to a variable of any pointer type.
• Function **malloc** is normally used with the **sizeof** operator.
• The memory allocated by **malloc** is not initialized.
• If no memory is available, **malloc** returns **NULL**.
• Function **free** deallocates memory so that the memory can be reallocated in the future.
• C also provides functions **calloc** and **realloc** for creating and modifying dynamic arrays.

Section 12.4 Linked Lists
• A linked list is a linear collection of self-referential structures, called nodes, connected by pointer links.
• A linked list is accessed via a pointer to the first node. Subsequent nodes are accessed via the link pointer member stored in each node.
• By convention, the link pointer in the last node of a list is set to **NULL** to mark the end of the list.
• Data is stored in a linked list dynamically—each node is created as necessary.
• A node can contain data of any type including other **struct** objects.
• Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
• Linked list nodes are normally not stored contiguously in memory. Logically, however, the nodes of a linked list appear to be contiguous.

Section 12.5 Stacks
• A stack is a constrained version of a linked list. New nodes can be added to a stack and removed from a stack only at the top—referred to as a last-in, first-out (LIFO) data structure.
• The primary functions used to manipulate a stack are **push** and **pop**. Function **push** creates a new node and places it on top of the stack. Function **pop** removes a node from the top of the stack, frees the memory that was allocated to the popped node and returns the popped value.
• Whenever a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack. If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order so that each function can return to its caller. Stacks support recursive function calls in the same manner as conventional nonrecursive calls.
• Stacks are used by compilers in the process of evaluating expressions and generating machine language code.
Section 12.6 Queues
• Queue nodes are removed only from the head of the queue and are inserted only at the tail of the queue—referred to as a first-in, first-out (FIFO) data structure.
• The insert and remove operations for a queue are known as enqueue and dequeue.

Section 12.7 Trees
• A tree is a nonlinear, two-dimensional data structure. Tree nodes contain two or more links.
• Binary trees are trees whose nodes all contain two links.
• The root node is the first node in a tree. Each link in the root node of a binary tree refers to a child. The left child is the first node in the left subtree, and the right child is the first node in the right subtree. The children of a node are called siblings.
• A node with no children is called a leaf node.
• A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node, and the values in any right subtree are greater than the value in its parent node.
• A node can only be inserted as a leaf node in a binary search tree.
• The steps for an in-order traversal are: Traverse the left subtree in-order, process the value in the node, then traverse the right subtree in-order. The value in a node is not processed until the values in its left subtree are processed.
• The in-order traversal of a binary search tree processes the node values in ascending order. The process of creating a binary search tree actually sorts the data—and thus this process is called the binary tree sort.
• The steps for a pre-order traversal are: Process the value in the node, traverse the left subtree pre-order, then traverse the right subtree pre-order. The value in each node is processed as the node is visited. After the value in a given node is processed, the values in the left subtree are processed, then the values in the right subtree are processed.
• The steps for a post-order traversal are: Traverse the left subtree post-order, traverse the right subtree post-order, then process the value in the node. The value in each node is not processed until the values of its children are processed.
• A binary search tree facilitates duplicate elimination. As the tree is being created, an attempt to insert a duplicate value will be recognized because a duplicate will follow the same “go left” or “go right” decisions on each comparison as the original value did. Thus, the duplicate will eventually be compared with a node in the tree containing the same value. The duplicate value may simply be discarded at this point.
• Searching a binary tree for a value that matches a key value is fast. If the tree is tightly packed, each level contains about twice as many elements as the previous level. So a binary search tree with \(n\) elements would have a maximum of \(\log_2 n\) levels, and thus a maximum of \(\log_2 n\) comparisons would have to be made either to find a match or to determine that no match exists. This means that when searching a (tightly packed) 1000-element binary search tree, no more than 10 comparisons need to be made because \(2^{10} > 1000\). When searching a (tightly packed) 1,000,000-element binary search tree, no more than 20 comparisons need to be made because \(2^{20} > 1,000,000\).
dynamic memory allocation 456
enqueue function of a queue 473
first-in first-out (FIFO) 472
free function 456
head of a queue 455
infix notation 489
inorder 479
last-in-first-out (LIFO) 466
leaf node 478
left child 478
left subtree 478
level order binary tree traversal 493
linear data structures 478
link 458
link (pointer in a self-referential structure) 456
linked list 455, 458
malloc function 456
node 458
NULL pointer 456
parent node 479
pointer to pointer (double indirection) 463
pointer to void (void *) 456
postfix notation 489
postorder 479
predicate function 463
preorder 479
queue 455, 472
replacement node 493
right child 478
right subtree 478
root node of a binary tree 478
self-referential structure 456
siblings 478
sizeof operator 456
stack 455, 466
tail of a queue 455
top of a stack 455
tree 478

Self-Review Exercises

12.1 Fill in the blanks in each of the following:
   a) A self-________ structure is used to form dynamic data structures.
   b) Function ________ is used to dynamically allocate memory.
   c) A(n) ________ is a specialized version of a linked list in which nodes can be inserted and deleted only from the start of the list.
   d) Functions that look at a linked list but do not modify it are referred to as ________.
   e) A queue is referred to as a(n) ________ data structure.
   f) The pointer to the next node in a linked list is referred to as a(n) ________.
   g) Function ________ is used to reclaim dynamically allocated memory.
   h) A(n) ________ is a specialized version of a linked list in which nodes can be inserted only at the start of the list and deleted only from the end of the list.
   i) A(n) ________ is a nonlinear, two-dimensional data structure that contains nodes with two or more links.
   j) A stack is referred to as a(n) ________ data structure because the last node inserted is the first node removed.
   k) The nodes of a(n) ________ tree contain two link members.
   l) The first node of a tree is the ________ node.
   m) Each link in a tree node points to a(n) ________ or ________ of that node.
   n) A tree node that has no children is called a(n) ________ node.
   o) The three traversal algorithms (covered in this chapter) for a binary tree are ________, ________ and ________.

12.2 What are the differences between a linked list and a stack?

12.3 What are the differences between a stack and a queue?

12.4 Write a statement or set of statements to accomplish each of the following. Assume that all the manipulations occur in main (therefore, no addresses of pointer variables are needed), and assume the following definitions:

```c
struct gradeNode {
    char lastName[20];
};
```
Answers to Self-Review Exercises

12.1  a) referential.  b) malloc.  c) stack.  d) predicates.  e) FIFO.  f) link.  g) free.  h) queue.  i) tree.  j) LIFO.  k) binary.  l) root.  m) child, subtree.  n) leaf.  o) inorder, preorder postorder.

12.2  It’s possible to insert a node anywhere in a linked list, and remove a node from anywhere in a linked list. However, nodes in a stack may only be inserted at the top of the stack and removed from the top of a stack.

12.3  A queue has pointers to both its head and its tail so that nodes may be inserted at the tail and deleted from the head. A stack has a single pointer to the top of the stack where both insertion and deletion of nodes is performed.

12.4  a) GradeNodePtr startPtr = NULL;
b) GradeNodePtr newPtr;
   newPtr = malloc( sizeof( GradeNode ) );
```c
strcpy(newPtr->lastName, "Jones");
newPtr->grade = 91.5;
newPtr->nextPtr = NULL;
c) To insert "Adams":
    previousPtr is NULL, currentPtr points to the first element in the list.
    newPtr->nextPtr = currentPtr;
    startPtr = newPtr;
To insert "Thompson":
    previousPtr points to the last element in the list (containing "Smith")
    currentPtr is NULL.
    newPtr->nextPtr = currentPtr;
    previousPtr->nextPtr = newPtr;
To insert "Pritchard":
    previousPtr points to the node containing "Jones"
    currentPtr points to the node containing "Smith"
    newPtr->nextPtr = currentPtr;
    previousPtr->nextPtr = newPtr;
d) currentPtr = startPtr;
    while (currentPtr != NULL) {
        printf("Lastname = %s\nGrade = %6.2f\n", 
            currentPtr->lastName, currentPtr->grade);
        currentPtr = currentPtr->nextPtr;
    }
    while (currentPtr != NULL) {
        tempPtr = currentPtr;
        currentPtr = currentPtr->nextPtr;
        free(tempPtr);
    }
    startPtr = NULL;
12.5 The inorder traversal is:
    11 18 19 28 32 40 44 49 69 72 83 92 97 99
The preorder traversal is:
    49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
The postorder traversal is:
    11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

Exercises
12.6 (Concatenating Lists) Write a program that concatenates two linked lists of characters. The
program should include function concatenate that takes pointers to both lists as arguments and
concatenates the second list to the first list.
12.7 (Merging Ordered Lists) Write a program that merges two ordered lists of integers into a
single ordered list of integers. Function merge should receive pointers to the first node of each of the
lists to be merged and should return a pointer to the first node of the merged list.
12.8 (Inserting into an Ordered List) Write a program that inserts 25 random integers from 0 to
100 in order in a linked list. The program should calculate the sum of the elements and the floating-
point average of the elements.
12.9  (Creating a Linked List, then Reversing Its Elements) Write a program that creates a linked list of 10 characters, then creates a copy of the list in reverse order.

12.10  (Reversing the Words of a Sentence) Write a program that inputs a line of text and uses a stack to print the line reversed.

12.11  (Palindrome Tester) Write a program that uses a stack to determine if a string is a palindrome (i.e., the string is spelled identically backward and forward). The program should ignore spaces and punctuation.

12.12  (Infix-to-Postfix Converter) Stacks are used by compilers to help in the process of evaluating expressions and generating machine language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of constants, operators and parentheses.

Humans generally write expressions like $3 + 4$ and $7 / 9$ in which the operator (+ or / here) is written between its operands—this is called infix notation. Computers “prefer” postfix notation in which the operator is written to the right of its two operands. The preceding infix expressions would appear in postfix notation as $3 4 +$ and $7 9 /$, respectively.

To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation, and then evaluate the postfix version of the expression. Each of these algorithms requires only a single left-to-right pass of the expression. Each algorithm uses a stack in support of its operation, and in each the stack is used for a different purpose.

In this exercise, you’ll write a version of the infix-to-postfix conversion algorithm. In the next exercise, you’ll write a version of the postfix expression evaluation algorithm.

Write a program that converts an ordinary infix arithmetic expression (assume a valid expression is entered) with single digit integers such as

$$(6 + 2) * 5 - 8 / 4$$

to a postfix expression. The postfix version of the preceding infix expression is

$$6 2 + 5 * 8 4 / -$$

The program should read the expression into character array $infix$, and use modified versions of the stack functions implemented in this chapter to help create the postfix expression in character array $postfix$. The algorithm for creating a postfix expression is as follows:

1) Push a left parenthesis '(' onto the stack.
2) Append a right parenthesis ')' to the end of $infix$.
3) While the stack is not empty, read $infix$ from left to right and do the following:
   - If the current character in $infix$ is a digit, copy it to the next element of $postfix$.
   - If the current character in $infix$ is a left parenthesis, push it onto the stack.
   - If the current character in $infix$ is an operator,
     - Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and insert the popped operators in $postfix$.
     - Push the current character in $infix$ onto the stack.
   - If the current character in $infix$ is a right parenthesis
     - Pop operators from the top of the stack and insert them in $postfix$ until a left parenthesis is at the top of the stack.
     - Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

+  addition
-  subtraction
*  multiplication
/  division
Chapter 12  C Data Structures

^  exponentiation
%  remainder
The stack should be maintained with the following declarations:

```c
struct stackNode {
    char data;
    struct stackNode *nextPtr;
};
typedef struct stackNode StackNode;
typedef StackNode *StackNodePtr;
```

The program should consist of main and eight other functions with the following function headers:

```c
void convertToPostfix( char infix[], char postfix[] )
Convert the infix expression to postfix notation.

int isOperator( char c )
Determine if c is an operator.

int precedence( char operator1, char operator2 )
Determine if the precedence of operator1 is less than, equal to, or greater than the precedence of operator2. The function returns -1, 0 and 1, respectively.

void push( StackNodePtr *topPtr, char value )
Push a value on the stack.

char pop( StackNodePtr *topPtr )
Pop a value off the stack.

char stackTop( StackNodePtr topPtr )
Return the top value of the stack without popping the stack.

int isEmpty( StackNodePtr topPtr )
Determine if the stack is empty.

void printStack( StackNodePtr topPtr )
Print the stack.
```

### 12.13 (Postfix Evaluator)
Write a program that evaluates a postfix expression (assume it’s valid) such as

```
6 2 + 5 * 8 4 / -
```

The program should read a postfix expression consisting of single digits and operators into a character array. Using modified versions of the stack functions implemented earlier in this chapter, the program should scan the expression and evaluate it. The algorithm is as follows:

1) Append the null character (‘\0’) to the end of the postfix expression. When the null character is encountered, no further processing is necessary.
2) While ‘\0’ has not been encountered, read the expression from left to right.
   If the current character is a digit,
   Push its integer value onto the stack (the integer value of a digit character is its value in the computer’s character set minus the value of ‘0’ in the computer’s character set).
   Otherwise, if the current character is an operator,
   Pop the two top elements of the stack into variables x and y.
   Calculate y operator x.
   Push the result of the calculation onto the stack.
3) When the null character is encountered in the expression, pop the top value of the stack.
   This is the result of the postfix expression.

[Note: In 2) above, if the operator is '/', the top of the stack is 2, and the next element in the stack
is 8, then pop 2 into x, pop 8 into y, evaluate 8 / 2, and push the result, 4, back on the stack. This
note also applies to operator '-'.]

The arithmetic operations allowed in an expression are:

+   addition
-   subtraction
*   multiplication
/   division
^   exponentiation
%   remainder

The stack should be maintained with the following declarations:

```
struct stackNode {
   int data;
   struct stackNode *nextPtr;
};

typedef struct stackNode StackNode;
typedef StackNode *StackNodePtr;
```

The program should consist of main and six other functions with the following function headers:

```
int evaluatePostfixExpression( char *expr )
   Evaluate the postfix expression.
int calculate( int op1, int op2, char operator )
   Evaluate the expression op1 operator op2.
void push( StackNodePtr *topPtr, int value )
   Push a value on the stack.
int pop( StackNodePtr *topPtr )
   Pop a value off the stack.
int isEmpty( StackNodePtr topPtr )
   Determine if the stack is empty.
void printStack( StackNodePtr topPtr )
   Print the stack.
```

12.14 (Postfix Evaluator Modification) Modify the postfix evaluator program of Exercise 12.13
so that it can process integer operands larger than 9.

12.15 (Supermarket Simulation) Write a program that simulates a check-out line at a supermar-
ket. The line is a queue. Customers arrive in random integer intervals of 1 to 4 minutes. Also, each
customer is serviced in random integer intervals of 1 to 4 minutes. Obviously, the rates need to be
balanced. If the average arrival rate is larger than the average service rate, the queue will grow in-
finitely. Even with balanced rates, randomness can still cause long lines. Run the supermarket simu-
lation for a 12-hour day (720 minutes) using the following algorithm:

1) Choose a random integer between 1 and 4 to determine the minute at which the first
customer arrives.
2) At the first customer’s arrival time:
   Determine customer’s service time (random integer from 1 to 4);
   Begin servicing the customer;
   Schedule arrival time of next customer (random integer 1 to 4 added to the current time).
3) For each minute of the day:
   If the next customer arrives,
       Say so;
       Enqueue the customer;
       Schedule the arrival time of the next customer;
   If service was completed for the last customer;
       Say so;
       Dequeue next customer to be serviced;
       Determine customer’s service completion time
           (random integer from 1 to 4 added to the current time).

Now run your simulation for 720 minutes and answer each of the following:
   a) What is the maximum number of customers in the queue at any time?
   b) What is the longest wait any one customer experienced?
   c) What happens if the arrival interval is changed from 1 to 4 minutes to 1 to 3 minutes?

12.16 (Allowing Duplicates in a Binary Tree) Modify the program of Fig. 12.19 to allow the binary tree to contain duplicate values.

12.17 (Binary Search Tree of Strings) Write a program based on the program of Fig. 12.19 that inputs a line of text, tokenizes the sentence into separate words, inserts the words in a binary search tree, and prints the inorder, preorder, and postorder traversals of the tree.
   [Hint: Read the line of text into an array. Use `strtok` to tokenize the text. When a token is found, create a new node for the tree, assign the pointer returned by `strtok` to member `string` of the new node, and insert the node in the tree.]

12.18 (Duplicate Elimination) In this chapter, we saw that duplicate elimination is straightforward when creating a binary search tree. Describe how you would perform duplicate elimination using only a single subscripted array. Compare the performance of array-based duplicate elimination with the performance of binary-search-tree-based duplicate elimination.

12.19 (Depth of a Binary Tree) Write a function `depth` that receives a binary tree and determines how many levels it has.

12.20 (Recursively Print a List Backwards) Write a function `printListBackwards` that recursively outputs the items in a list in reverse order. Use your function in a test program that creates a sorted list of integers and prints the list in reverse order.

12.21 (Recursively Search a List) Write a function `searchList` that recursively searches a linked list for a specified value. The function should return a pointer to the value if it’s found; otherwise, `NULL` should be returned. Use your function in a test program that creates a list of integers. The program should prompt the user for a value to locate in the list.

12.22 (Binary Tree Delete) In this exercise, we discuss deleting items from binary search trees. The deletion algorithm is not as straightforward as the insertion algorithm. There are three cases that are encountered when deleting an item—the item is contained in a leaf node (i.e., it has no children), the item is contained in a node that has one child, or the item is contained in a node that has two children.

   If the item to be deleted is contained in a leaf node, the node is deleted and the pointer in the parent node is set to `NULL`.

   If the item to be deleted is contained in a node with one child, the pointer in the parent node is set to point to the child node and the node containing the data item is deleted. This causes the child node to take the place of the deleted node in the tree.

   The last case is the most difficult. When a node with two children is deleted, another node must take its place. However, the pointer in the parent node cannot simply be assigned to point to one of the children of the node to be deleted. In most cases, the resulting binary search tree would
not adhere to the following characteristic of binary search trees: *The values in any left subtree are less than the value in the parent node, and the values in any right subtree are greater than the value in the parent node.*

Which node is used as a replacement node to maintain this characteristic? Either the node containing the largest value in the tree less than the value in the node being deleted, or the node containing the smallest value in the tree greater than the value in the node being deleted. Let's consider the node with the smaller value. In a binary search tree, the largest value less than a parent's value is located in the left subtree of the parent node and is guaranteed to be contained in the rightmost node of the subtree. This node is located by walking down the left subtree to the right until the pointer to the right child of the current node is `NULL`. We're now pointing to the replacement node which is either a leaf node or a node with one child to its left. If the replacement node is a leaf node, the steps to perform the deletion are as follows:

1) Store the pointer to the node to be deleted in a temporary pointer variable (this pointer is used to delete the dynamically allocated memory).
2) Set the pointer in the parent of the node being deleted to point to the replacement node.
3) Set the pointer in the parent of the replacement node to null.
4) Set the pointer to the right subtree in the replacement node to point to the right subtree of the node to be deleted.
5) Delete the node to which the temporary pointer variable points.

The deletion steps for a replacement node with a left child are similar to those for a replacement node with no children, but the algorithm also must move the child to the replacement node's position. If the replacement node is a node with a left child, the steps to perform the deletion are as follows:

1) Store the pointer to the node to be deleted in a temporary pointer variable.
2) Set the pointer in the parent of the node being deleted to point to the replacement node.
3) Set the pointer in the parent of the replacement node to point to the left child of the replacement node.
4) Set the pointer to the right subtree in the replacement node to point to the right subtree of the node to be deleted.
5) Delete the node to which the temporary pointer variable points.

Write function `deleteNode` which takes as its arguments a pointer to the root node of the tree and the value to be deleted. The function should locate in the tree the node containing the value to be deleted and use the algorithms discussed here to delete the node. If the value is not found in the tree, the function should print a message that indicates whether or not the value is deleted. Modify the program of Fig. 12.19 to use this function. After deleting an item, call the `inOrder`, `preOrder` and `postOrder` traversal functions to confirm that the delete operation was performed correctly.

**12.23 (Binary Tree Search)** Write function `binaryTreeSearch` that attempts to locate a specified value in a binary search tree. The function should take as arguments a pointer to the root node of the binary tree and a search key to be located. If the node containing the search key is found, the function should return a pointer to that node; otherwise, the function should return a `NULL` pointer.

**12.24 (Level Order Binary Tree Traversal)** The program of Fig. 12.19 illustrated three recursive methods of traversing a binary tree—inorder traversal, preorder traversal, and postorder traversal. This exercise presents the *level order traversal* of a binary tree in which the node values are printed level-by-level starting at the root node level. The nodes on each level are printed from left to right. The level order traversal is not a recursive algorithm. It uses the queue data structure to control the output of the nodes. The algorithm is as follows:

1) Insert the root node in the queue
2) While there are nodes left in the queue,
   Get the next node in the queue
   Print the node's value
If the pointer to the left child of the node is not null
  Insert the left child node in the queue
If the pointer to the right child of the node is not null
  Insert the right child node in the queue.

Write function `levelOrder` to perform a level order traversal of a binary tree. The function should take as an argument a pointer to the root node of the binary tree. Modify the program of Fig. 12.19 to use this function. Compare the output from this function to the outputs of the other traversal algorithms to see that it worked correctly. [Note: You’ll also need to modify and incorporate the queue processing functions of Fig. 12.13 in this program.]

12.25 (Printing Trees) Write a recursive function `outputTree` to display a binary tree on the screen. The function should output the tree row-by-row with the top of the tree at the left of the screen and the bottom of the tree toward the right of the screen. Each row is output vertically. For example, the binary tree illustrated in Fig. 12.22 is output as follows:

```
   99
  92  83
  72  71
  69  49
  44  40
  32
 19
 18
 11
```

Note the rightmost leaf node appears at the top of the output in the rightmost column, and the root node appears at the left of the output. Each column of output starts five spaces to the right of the previous column. Function `outputTree` should receive as arguments a pointer to the root node of the tree and an integer `totalSpaces` representing the number of spaces preceding the value to be output (this variable should start at zero so the root node is output at the left of the screen). The function uses a modified inorder traversal to output the tree—it starts at the rightmost node in the tree and works back to the left. The algorithm is as follows:

While the pointer to the current node is not null
  Recursively call `outputTree` with the current node’s right subtree and `totalSpaces + 5`
  Use a for statement to count from 1 to `totalSpaces` and output spaces
  Output the value in the current node
  Set the pointer to the current node to point to the left subtree of the current node
  Increment `totalSpaces` by 5.

Special Section: Building Your Own Compiler

In Exercises 7.27–7.29, we introduced Simpletron Machine Language (SML), and you implemented a Simpletron computer simulator to execute SML programs. In Exercises 12.26–12.30, we build a compiler that converts programs written in a high-level programming language to SML. This section “ties” together the entire programming process. You’ll write programs in this new high-level language, compile them on the compiler you build and run them on the simulator you built in Exercise 7.28. You should make every effort to implement your compiler in an object-oriented manner. [Note: Due to the size of the descriptions for Exercises 12.26–12.30, we’ve posted them in a PDF document located at www.deitel.com/books/chtp6/.]
Hold thou the good; define it well.
—Alfred, Lord Tennyson

I have found you an argument; but I am not obliged to find you an understanding.
—Samuel Johnson

A good symbol is the best argument, and is a missionary to persuade thousands.
—Ralph Waldo Emerson

The partisan, when he is engaged in a dispute, cares nothing about the rights of the question, but is anxious only to convince his hearers of his own assertions.
—Plato

**Objectives**

In this chapter, you’ll learn:

- To use `#include` to develop large programs.
- To use `#define` to create macros and macros with arguments.
- To use conditional compilation to specify portions of a program that should not always be compiled (such as code that assists you in debugging).
- To display error messages during conditional compilation.
- To use assertions to test if the values of expressions are correct.
Chapter 13  C Preprocessor

13.1 Introduction
The C preprocessor executes before a program is compiled. Some actions it performs are the inclusion of other files in the file being compiled, definition of symbolic constants and macros, conditional compilation of program code and conditional execution of preprocessor directives. Preprocessor directives begin with # and only white-space characters and comments may appear before a preprocessor directive on a line.

13.2 #include Preprocessor Directive
The #include preprocessor directive has been used throughout this text. The #include directive causes a copy of a specified file to be included in place of the directive. The two forms of the #include directive are:

```
#include <filename>
#include "Filename"
```

The difference between these is the location the preprocessor begins searches for the file to be included. If the file name is enclosed in quotes, the preprocessor starts searches in the same directory as the file being compiled for the file to be included (and may search other locations as well). This method is normally used to include programmer-defined headers. If the file name is enclosed in angle brackets (< and >)—used for standard library headers—the search is performed in an implementation-dependent manner, normally through predesignated compiler and system directories.

The #include directive is used to include standard library headers such as stdio.h and stdlib.h (see Fig. 5.6) and with programs consisting of several source files that are to be compiled together. A header containing declarations common to the separate program files is often created and included in the file. Examples of such declarations are structure and union declarations, enumerations and function prototypes.

13.3 #define Preprocessor Directive: Symbolic Constants
The #define directive creates symbolic constants—constants represented as symbols—and macros—operations defined as symbols. The #define directive format is

```
#define identifier replacement-text
```
When this line appears in a file, all subsequent occurrences of identifier that do not appear in string literals will be replaced by replacement-text automatically before the program is compiled. For example,

```c
#define PI 3.14159
```

replaces all subsequent occurrences of the symbolic constant PI with the numeric constant 3.14159. Symbolic constants enable you to create a name for a constant and use the name throughout the program. If the constant needs to be modified throughout the program, it can be modified once in the `#define` directive. When the program is recompiled, all occurrences of the constant in the program will be modified accordingly. [Note: Everything to the right of the symbolic constant name replaces the symbolic constant.] For example, `#define PI = 3.14159` causes the preprocessor to replace every occurrence of the identifier PI with = 3.14159. This is the cause of many subtle logic and syntax errors. Redefining a symbolic constant with a new value is also an error.

### Good Programming Practice 13.1
Using meaningful names for symbolic constants helps make programs more self-documenting.

### Good Programming Practice 13.2
By convention, symbolic constants are defined using only uppercase letters and underscores.

### 13.4 `#define` Preprocessor Directive: Macros

A macro is an identifier defined in a `#define` preprocessor directive. As with symbolic constants, the macro-identifier is replaced in the program with the replacement-text before the program is compiled. Macros may be defined with or without arguments. A macro without arguments is processed like a symbolic constant. In a macro with arguments, the arguments are substituted in the replacement text, then the macro is expanded—i.e., the replacement-text replaces the identifier and argument list in the program. [Note: A symbolic constant is a type of macro.]

Consider the following macro definition with one argument for the area of a circle:

```c
#define CIRCLE_AREA( x ) ( ( PI ) * ( x ) * ( x ) )
```

Wherever `CIRCLE_AREA(y)` appears in the file, the value of y is substituted for x in the replacement-text, the symbolic constant PI is replaced by its value (defined previously) and the macro is expanded in the program. For example, the statement

```c
area = CIRCLE_AREA( 4 );
```

is expanded to

```c
area = ( ( 3.14159 ) * ( 4 ) * ( 4 ) );
```

and the value of the expression is evaluated and assigned to variable area. The parentheses around each x in the replacement text force the proper order of evaluation when the macro argument is an expression. For example, the statement

```c
area = CIRCLE_AREA( c + 2 );
```
Chapter 13  C Preprocessor

is expanded to

\[
\text{area} = ( ( 3.14159 ) \times ( c + 2 ) \times ( c + 2 ) );
\]

which evaluates correctly because the parentheses force the proper order of evaluation. If the parentheses are omitted, the macro expansion is

\[
\text{area} = 3.14159 \times c + 2 \times c + 2;
\]

which evaluates incorrectly as

\[
\text{area} = ( 3.14159 \times c ) + ( 2 \times c ) + 2;
\]

because of the rules of operator precedence.

**Common Programming Error 13.1**

Forgetting to enclose macro arguments in parentheses in the replacement text can lead to logic errors.

Macro CIRCLE_AREA could be defined as a function. Function circleArea

\[
\begin{align*}
\text{double circleArea( double x )} \\
& \{ \\
& \quad \text{return 3.14159} \times x \times x; \\
& \}
\end{align*}
\]

performs the same calculation as macro CIRCLE_AREA, but the overhead of a function call is associated with function circleArea. The advantages of macro CIRCLE_AREA are that macros insert code directly in the program—avoiding function call overhead—and the program remains readable because the CIRCLE_AREA calculation is defined separately and named meaningfully. A disadvantage is that its argument is evaluated twice.

**Performance Tip 13.1**

Macros can sometimes be used to replace a function call with inline code to eliminate the overhead of a function call. Today’s optimizing compilers often inline functions for you, so many programmers no longer use macros for this purpose. C99 also provides the inline keyword (see Appendix G).

The following is a macro definition with two arguments for the area of a rectangle:

\[
\text{#define RECTANGLE_AREA( x, y ) \{ ( x ) \times ( y ) \}}
\]

Wherever RECTANGLE_AREA(x, y) appears in the program, the values of x and y are substituted in the macro replacement text and the macro is expanded in place of the macro name. For example, the statement

\[
\text{rectArea} = \text{RECTANGLE_AREA( a + 4, b + 7 );}
\]

is expanded to

\[
\text{rectArea} = ( ( a + 4 ) \times ( b + 7 ) );
\]

The value of the expression is evaluated and assigned to variable rectArea.

The replacement text for a macro or symbolic constant is normally any text on the line after the identifier in the #define directive. If the replacement text for a macro or symbolic constant is longer than the remainder of the line, a backslash (\) must be placed at the end of the line, indicating that the replacement text continues on the next line.
Symbolic constants and macros can be discarded by using the \#undef preprocessor directive. Directive \#undef “undefines” a symbolic constant or macro name. The scope of a symbolic constant or macro is from its definition until it is undefined with \#undef, or until the end of the file. Once undefined, a name can be redefined with \#define.

Functions in the standard library sometimes are defined as macros based on other library functions. A macro commonly defined in the stdio.h header is

```c
#define getchar() getc( stdin )
```

The macro definition of getchar uses function getc to get one character from the standard input stream. Function putchar of the stdio.h header and the character handling functions of the ctype.h header often are implemented as macros as well. Expressions with side effects (i.e., variable values are modified) should not be passed to a macro because macro arguments may be evaluated more than once.

### 13.5 Conditional Compilation

Conditional compilation enables you to control the execution of preprocessor directives and the compilation of program code. Each of the conditional preprocessor directives evaluates a constant integer expression. Cast expressions, sizeof expressions and enumeration constants cannot be evaluated in preprocessor directives.

The conditional preprocessor construct is much like the if selection statement. Consider the following preprocessor code:

```c
#if !defined(MY_CONSTANT)
#define MY_CONSTANT 0
#endif
```

These directives determine if MY_CONSTANT is defined. The expression defined(MY_CONSTANT) evaluates to 1 if MY_CONSTANT is defined; 0 otherwise. If the result is 0, !defined(MY_CONSTANT) evaluates to 1 and MY_CONSTANT is defined. Otherwise, the \#define directive is skipped. Every \#if construct ends with \#endif. Directives \#ifdef and \#ifndef are shorthand for \#if defined(name) and \#if !defined(name). A multiple-part conditional preprocessor construct may be tested by using the \#elif (the equivalent of else if in an if statement) and the \#else (the equivalent of else in an if statement) directives. These directives are frequently used to prevent header files from being included multiple times in the same source file. We use this technique extensively in the C++ part of this book.

During program development, it is often helpful to “comment out” portions of code to prevent it from being compiled. If the code contains comments, /* and */ cannot be used to accomplish this task. Instead, you can use the following preprocessor construct:

```c
#if 0
code prevented from compiling
#endif
```

To enable the code to be compiled, replace the 0 in the preceding construct with 1.

Conditional compilation is commonly used as a debugging aid. Many C implementations provide debuggers, which provide much more powerful features than conditional compilation. If a debugger is not available, printf statements are often used to print variable values and to confirm the flow of control. These printf statements can be enclosed
in conditional preprocessor directives so the statements are only compiled while the debugging process is not completed. For example,

```c
#ifdef DEBUG
    printf("Variable x = %d\n", x);
#endif
```

causes a `printf` statement to be compiled in the program if the symbolic constant `DEBUG` has been defined (`#define DEBUG`) before directive `#ifdef DEBUG`. When debugging is completed, the `#define` directive is removed from the source file (or commented out) and the `printf` statements inserted for debugging purposes are ignored during compilation. In larger programs, it may be desirable to define several different symbolic constants that control the conditional compilation in separate sections of the source file.

### Common Programming Error 13.2

Inserting conditionally compiled `printf` statements for debugging purposes in locations where C currently expects a single statement. In this case, the conditionally compiled statement should be enclosed in a compound statement. Thus, when the program is compiled with debugging statements, the flow of control of the program is not altered.

#### 13.6 `#error` and `#pragma` Preprocessor Directives

The `#error` directive

```c
#error tokens
```

prints an implementation-dependent message including the tokens specified in the directive. The tokens are sequences of characters separated by spaces. For example,

```c
#error 1 - Out of range error
```

contains 6 tokens. When a `#error` directive is processed on some systems, the tokens in the directive are displayed as an error message, preprocessing stops and the program does not compile.

The `#pragma` directive

```c
#pragma tokens
```

causes an implementation-defined action. A pragma not recognized by the implementation is ignored. For more information on `#error` and `#pragma`, see the documentation for your C implementation.

#### 13.7 `#` and `##` Operators

The `#` and `##` preprocessor operators are available in Standard C. The `#` operator causes a replacement text token to be converted to a string surrounded by quotes. Consider the following macro definition:

```c
#define HELLO(x) printf("Hello, " #x "\n");
```

When `HELLO(John)` appears in a program file, it is expanded to

```c
printf("Hello, " "John" "\n");
```
The string "John" replaces #x in the replacement text. Strings separated by white space are concatenated during preprocessing, so the preceding statement is equivalent to

```
printf( "Hello, John\\n" );
```

The # operator must be used in a macro with arguments because the operand of # refers to an argument of the macro.

The ## operator concatenates two tokens. Consider the following macro definition:

```
#define TOKENCONCAT(x, y)  x ## y
```

When TOKENCONCAT appears in the program, its arguments are concatenated and used to replace the macro. For example, `TOKENCONCAT(0, K)` is replaced by `0K` in the program. The ## operator must have two operands.

### 13.8 Line Numbers

The `#line` preprocessor directive causes the subsequent source code lines to be renumbered starting with the specified constant integer value. The directive

```c
#line 100
```

starts line numbering from 100 beginning with the next source code line. A file name can be included in the `#line` directive. The directive

```c
#line 100 "file1.c"
```

indicates that lines are numbered from 100 beginning with the next source code line and that the name of the file for the purpose of any compiler messages is "file1.c". The directive normally is used to help make the messages produced by syntax errors and compiler warnings more meaningful. The line numbers do not appear in the source file.

### 13.9 Predefined Symbolic Constants

Standard C provides predefined symbolic constants, several of which are shown in Fig. 13.1. The identifiers for each of the predefined symbolic constants begin and end with two underscores. These identifiers and the `defined` identifier (used in Section 13.5) cannot be used in `#define` or `#undef` directives.

<table>
<thead>
<tr>
<th>Symbolic constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LINE</strong></td>
<td>The line number of the current source code line (an integer constant).</td>
</tr>
<tr>
<td><strong>FILE</strong></td>
<td>The presumed name of the source file (a string).</td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>The date the source file was compiled (a string of the form &quot;Mmm dd yyyy&quot; such as &quot;Jan 19 2002&quot;).</td>
</tr>
<tr>
<td><strong>TIME</strong></td>
<td>The time the source file was compiled (a string literal of the form &quot;hh:mm:ss&quot;).</td>
</tr>
<tr>
<td><strong>STDC</strong></td>
<td>The value 1 if the compiler supports Standard C.</td>
</tr>
</tbody>
</table>

**Fig. 13.1** | Some predefined symbolic constants.
13.10 Assertions

The `assert` macro—defined in the `<assert.h>` header—tests the value of an expression. If the value of the expression is false (0), `assert` prints an error message and calls function `abort` (of the general utilities library—`<stdlib.h>`) to terminate program execution. This is a useful debugging tool for testing if a variable has a correct value. For example, suppose variable `x` should never be larger than 10 in a program. An assertion may be used to test the value of `x` and print an error message if the value of `x` is incorrect. The statement would be

```c
assert( x <= 10 );
```

If `x` is greater than 10 when the preceding statement is encountered in a program, an error message containing the line number and file name is printed and the program terminates. You may then concentrate on this area of the code to find the error. If the symbolic constant `NDEBUG` is defined, subsequent assertions will be ignored. Thus, when assertions are no longer needed, the line

```c
#define NDEBUG
```

is inserted in the program file rather than deleting each assertion manually.

Software Engineering Observation 13.1

Assertions are not meant as a substitute for error handling during normal runtime conditions. Their use should be limited to finding logic errors.

Summary

Section 13.1 Introduction

• The preprocessor executes before a program is compiled.
• All preprocessor directives begin with `#`.

Section 13.2 `#include` Preprocessor Directive

• Only white-space characters and comments may appear before a preprocessor directive on a line.
• The `#include` directive includes a copy of the specified file. If the file name is enclosed in quotes, the preprocessor begins searching in the same directory as the file being compiled for the file to be included. If the file name is enclosed in angle brackets (`<` and `>`), the search is performed in an implementation-defined manner.

Section 13.3 `#define` Preprocessor Directive: Symbolic Constants

• The `#define` preprocessor directive is used to create symbolic constants and macros.
• A symbolic constant is a name for a constant.
• A macro is an operation defined in a `#define` preprocessor directive. Macros may be defined with or without arguments.

Section 13.4 `#define` Preprocessor Directive: Macros

• The replacement text for a macro or symbolic constant is any text remaining on the line after the identifier in the `#define` directive. If the replacement text for a macro or symbolic constant is longer than the remainder of the line, a backslash (`\`) is placed at the end of the line indicating that the replacement text continues on the next line.
• Symbolic constants and macros can be discarded by using the \texttt{#undef} preprocessor directive. Directive \texttt{#undef} “undefines” the symbolic constant or macro name.
• The scope of a symbolic constant or macro is from its definition until it is undefined with \texttt{#undef} or until the end of the file.

Section 13.5 Conditional Compilation
• Conditional compilation enables you to control the execution of preprocessor directives and the compilation of program code.
• The conditional preprocessor directives evaluate constant integer expressions. Cast expressions, \texttt{sizeof} expressions and enumeration constants cannot be evaluated in preprocessor directives.
• Every \texttt{#if} construct ends with \texttt{#endif}.
• Directives \texttt{#ifdef} and \texttt{#ifndef} are provided as shorthand for \texttt{#if defined(name)} and \texttt{#if !defined(name)}.
• Multiple-part conditional preprocessor constructs may be tested with directives \texttt{#elif} and \texttt{#else}.

Section 13.6 \texttt{#error} and \texttt{#pragma} Preprocessor Directives
• The \texttt{#error} directive prints an implementation-dependent message that includes the tokens specified in the directive.
• The \texttt{#pragma} directive causes an implementation-defined action. If the pragma is not recognized by the implementation, the pragma is ignored.

Section 13.7 \texttt{#} and \texttt{##} Operators
• The \texttt{#} operator causes a replacement text token to be converted to a string surrounded by quotes. The \texttt{#} operator must be used in a macro with arguments, because the operand of \texttt{#} must be an argument of the macro.
• The \texttt{##} operator concatenates two tokens. The \texttt{##} operator must have two operands.

Section 13.8 Line Numbers
• The \texttt{#line} preprocessor directive causes the subsequent source code lines to be renumbered starting with the specified constant integer value.

Section 13.9 Predefined Symbolic Constants
• Constant \texttt{__LINE__} is the line number (an integer) of the current source code line. Constant \texttt{__FILE__} is the presumed name of the file (a string). Constant \texttt{__DATE__} is the date the source file is compiled (a string). Constant \texttt{__TIME__} is the time the source file is compiled (a string). Constant \texttt{__STDC__} indicates whether the compiler supports Standard C. Each of the predefined symbolic constants begins and ends with two underscores.

Section 13.10 Assertions
• Macro \texttt{assert}—defined in the \texttt{<assert.h>} header—tests the value of an expression. If the value of the expression is 0 (false), \texttt{assert} prints an error message and calls function \texttt{abort} to terminate program execution.

Terminology

\begin{tabular}{ll}
\texttt{abort} function & 502  \\
arguments & 497  \\
\texttt{assert macro} & 502  \\
\texttt{<assert.h>} & 502  \\
backslash () & 498  \\
C preprocessor & 496  \\
conditional compilation & 496  \\
conditional execution of preprocessor directive & 496  \\
debugger & 499  \\
\texttt{#define} preprocessor directive & 496  \\
\texttt{#elif} preprocessor directive & 499  \\
\end{tabular}
Self-Review Exercises

13.1 Fill in the blanks in each of the following:
   a) Every preprocessor directive must begin with _______.
   b) The conditional compilation construct may be extended to test for multiple cases by using the _______ and the _______ directives.
   c) The _______ directive creates macros and symbolic constants.
   d) Only _______ characters may appear before a preprocessor directive on a line.
   e) The _______ directive discards symbolic constant and macro names.
   f) The _______ and _______ directives are provided as shorthand notation for #if defined(name) and #if !defined(name).
   g) _______ enables you to control the execution of preprocessor directives and the compilation of program code.
   h) The _______ macro prints a message and terminates program execution if the value of the expression the macro evaluates is 0.
   i) The _______ directive inserts a file in another file.
   j) The _______ operator concatenates its two arguments.
   k) The _______ operator converts its operand to a string.
   l) The character _______ indicates that the replacement text for a symbolic constant or macro continues on the next line.
   m) The _______ directive causes the source code lines to be numbered from the indicated value beginning with the next source code line.

13.2 Write a program to print the values of the predefined symbolic constants listed in Fig. 13.1.

13.3 Write a preprocessor directive to accomplish each of the following:
   a) Define symbolic constant YES to have the value 1.
   b) Define symbolic constant NO to have the value 0.
   c) Include the header common.h. The header is found in the same directory as the file being compiled.
   d) Renumber the remaining lines in the file beginning with line number 3000.
   e) If symbolic constant TRUE is defined, undefine it and redefine it as 1. Do not use #ifdef.
   f) If symbolic constant TRUE is defined, undefine it and redefine it as 1. Use the #ifdef preprocessor directive.
   g) If symbolic constant TRUE is not equal to 0, define symbolic constant FALSE as 0. Otherwise define FALSE as 1.
   h) Define macro CUBE_VOLUME that computes the volume of a cube. The macro takes one argument.

Answers to Self-Review Exercises

13.1 a) #.  b) #elif, #else.  c) #define.  d) white-space.  e) #undef.  f) #ifdef, #ifndef.  g) Conditional compilation.  h) assert.  i) #include.  j) ##.  k) #.  l) \.  m) #line.
13.2 See below.

```c
/* Print the values of the predefined macros */
#include <stdio.h>
int main( void ) {
  printf( "LINE = %d\n", __LINE__ );
  printf( "FILE = %s\n", __FILE__ );
  printf( "DATE = %s\n", __DATE__ );
  printf( "TIME = %s\n", __TIME__ );
  printf( "STDC = %s\n", __STDC__ );
  return 0;
}
```

Exercises

13.3

a) `#define YES 1`
b) `#define NO 0`
c) `#include "common.h"`
d) `#line 3000`
e) `#if defined( TRUE )
   #undef TRUE
   #define TRUE 1
#endif`
f) `#ifdef TRUE
   #undef TRUE
   #define TRUE 1
#endif`
g) `#if TRUE
   #define FALSE 0
#else
   #define FALSE 1
#endif`
h) `#define CUBE_VOLUME( x ) ( ( x ) * ( x ) * ( x ) )`

Exercises

13.4 (Volume of a Sphere) Write a program that defines a macro with one argument to compute the volume of a sphere. The program should compute the volume for spheres of radius 1 to 10 and print the results in tabular format. The formula for the volume of a sphere is

\[
\left( \frac{4.0}{3} \right) \pi r^3
\]

where \( \pi \) is 3.14159.

13.5 (Adding Two Numbers) Write a program that produces the following output:

```
The sum of x and y is 13
```

The program should define macro SUM with two arguments, x and y, and use SUM to produce the output.
13.6 (Smallest of Two Numbers) Write a program that defines and uses macro MINIMUM2 to determine the smallest of two numeric values. Input the values from the keyboard.

13.7 (Smallest of Three Numbers) Write a program that defines and uses macro MINIMUM3 to determine the smallest of three numeric values. Macro MINIMUM3 should use macro MINIMUM2 defined in Exercise 13.6 to determine the smallest number. Input the values from the keyboard.

13.8 (Printing a String) Write a program that defines and uses macro PRINT to print a string value.

13.9 (Printing an Array) Write a program that defines and uses macro PRINTARRAY to print an array of integers. The macro should receive the array and the number of elements in the array as arguments.

13.10 (Totaling an Array's Contents) Write a program that defines and uses macro SUMARRAY to sum the values in a numeric array. The macro should receive the array and the number of elements in the array as arguments.
We’ll use a signal I have tried and found far-reaching and easy to yell. Waa-boo!
—Zane Grey

It is quite a three-pipe problem.
—Sir Arthur Conan Doyle

Objectives
In this chapter, you’ll learn:
- To redirect keyboard input to come from a file.
- To redirect screen output to be placed in a file.
- To write functions that use variable-length argument lists.
- To process command-line arguments.
- To assign specific types to numeric constants.
- To use temporary files.
- To process external asynchronous events in a program.
- To allocate memory dynamically for arrays.
- To change the size of memory that was dynamically allocated previously.
14.1 Introduction

This chapter presents several additional topics not ordinarily covered in introductory courses. Many of the capabilities discussed here are specific to particular operating systems, especially Linux/UNIX and Windows.

14.2 Redirecting I/O

Normally the input to a program is from the keyboard (standard input), and the output from a program is displayed on the screen (standard output). On most computer systems—Linux/UNIX and Windows systems in particular—it’s possible to redirect inputs to come from a file rather than the keyboard and redirect outputs to be placed in a file rather than on the screen. Both forms of redirection can be accomplished without using the file-processing capabilities of the standard library.

There are several ways to redirect input and output from the command line. Consider the executable file `sum` (on Linux/UNIX systems) that inputs integers one at a time and keeps a running total of the values until the end-of-file indicator is set, then prints the result. Normally the user inputs integers from the keyboard and enters the end-of-file key combination to indicate that no further values will be input. With input redirection, the input can be stored in a file. For example, if the data is stored in file `input`, the command line

```
$ sum < input
```

executes the program `sum`; the redirect input symbol (`<`) indicates that the data in file `input` is to be used as input by the program. Redirecting input on a Windows system is performed identically.

The character `$` is a typical Linux/UNIX command line prompt (some systems use a `>` prompt or other symbol). Students often find it difficult to understand that redirection is an operating system function, not another C feature.

The second method of redirecting input is piping. A pipe (|) causes the output of one program to be redirected as the input to another program. Suppose program `random` outputs a series of random integers; the output of `random` can be “piped” directly to program `sum` using the command line

```
$ random | sum
```

This causes the sum of the integers produced by `random` to be calculated. Piping is performed identically in Linux/UNIX and Windows.

Program output can be redirected to a file by using the `redirect output symbol (>)`. For example, to redirect the output of program `random` to file `out`, use

```
$ random > out
```

Finally, program output can be appended to the end of an existing file by using the `append output symbol (>>)`. For example, to append the output from program `random` to file `out` created in the preceding command line, use the command line

```
$ random >> out
```

## 14.3 Variable-Length Argument Lists

It’s possible to create functions that receive an unspecified number of arguments. Most programs in the text have used the standard library function `printf` which, as you know, takes a variable number of arguments. As a minimum, `printf` must receive a string as its first argument, but `printf` can receive any number of additional arguments. The function prototype for `printf` is

```c
int printf(const char *format, ...);
```

The ellipsis (...) in the function prototype indicates that the function receives a variable number of arguments of any type. The ellipsis must always be placed at the end of the parameter list.

The macros and definitions of the `variable arguments headers <stdarg.h>` (Fig. 14.1) provide the capabilities necessary to build functions with variable-length argument lists. Figure 14.2 demonstrates function `average` (lines 26–41) that receives a variable number of arguments. The first argument of `average` is always the number of values to be averaged.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>va_list</td>
<td>A type suitable for holding information needed by macros <code>va_start</code>, <code>va_arg</code> and <code>va_end</code>. To access the arguments in a variable-length argument list, an object of type <code>va_list</code> must be defined.</td>
</tr>
<tr>
<td>va_start</td>
<td>A macro that is invoked before the arguments of a variable-length argument list can be accessed. The macro initializes the object declared with <code>va_list</code> for use by the <code>va_arg</code> and <code>va_end</code> macros.</td>
</tr>
<tr>
<td>va_arg</td>
<td>A macro that expands to an expression of the value and type of the next argument in the variable-length argument list. Each invocation of <code>va_arg</code> modifies the object declared with <code>va_list</code> so that the object points to the next argument in the list.</td>
</tr>
<tr>
<td>va_end</td>
<td>A macro that facilitates a normal return from a function whose variable-length argument list was referred to by the <code>va_start</code> macro.</td>
</tr>
</tbody>
</table>

**Fig. 14.1** | `stdarg.h` variable-length argument list type and macros.
/* Fig. 14.2: fig14_02.c  
Using variable-length argument lists */
#include <stdio.h>
#include <stdarg.h>

double average( int i, ... ); /* prototype */

int main( void )
{
    double w = 37.5;
    double x = 22.5;
    double y = 1.7;
    double z = 10.2;

    printf( "w = %.1f
x = %.1f
y = %.1f
z = %.1f

", w, x, y, z );

    printf( "The average of w and x is %.3f
The average of w, x, and y is %.3f
The average of w, x, y, and z is %.3f

", average( 2, w, x ),
    average( 3, w, x, y ),
    average( 4, w, x, y, z ) );
    return 0; /* indicates successful termination */
} /* end main */

double average( int i, ... )
{
    double total = 0; /* initialize total */
    int j; /* counter for selecting arguments */

    va_list ap; /* stores information needed by va_start and va_end */
    va_start( ap, i ); /* initializes the va_list object */

    /* process variable length argument list */
    for ( j = 1; j <= i; j++ ) {
        total += va_arg( ap, double );
    } /* end for */

    va_end( ap ); /* clean up variable-length argument list */
    return total / i; /* calculate average */
} /* end function average */

w = 37.5
x = 22.5
y = 1.7
z = 10.2

The average of w and x is 30.000
The average of w, x, and y is 20.567
The average of w, x, y, and z is 17.975
Function average (lines 26–41) uses all the definitions and macros of header `<stdarg.h>`. Object ap, of type `va_list` (line 30), is used by macros `va_start`, `va_arg` and `va_end` to process the variable-length argument list of function average. The function begins by invoking macro `va_start` (line 32) to initialize object ap for use in `va_arg` and `va_end`. The macro receives two arguments—object ap and the identifier of the rightmost argument in the argument list before the ellipsis—in this case (`va_start` uses `i` here to determine where the variable-length argument list begins). Next function average repeatedly adds the arguments in the variable-length argument list to variable `total` (lines 37–39). The value to be added to `total` is retrieved from the argument list by invoking macro `va_arg`. Macro `va_arg` receives two arguments—object ap and the type of the value expected in the argument list—`double` in this case. The macro returns the value of the argument. Function average invokes macro `va_end` (line 39) with object ap as an argument to facilitate a normal return to `main` from `average`. Finally, the average is calculated and returned to `main`.

The reader may question how function `printf` and function `scanf` know what type to use in each `va_arg` macro. The answer is that `printf` and `scanf` scan the format conversion specifiers in the format control string to determine the type of the next argument to be processed.

### 14.4 Using Command-Line Arguments

On many systems, it’s possible to pass arguments to `main` from a command line by including parameters `int argc` and `char *argv[]` in the parameter list of `main`. Parameter `argc` receives the number of command-line arguments. Parameter `argv` is an array of strings in which the actual command-line arguments are stored. Common uses of command-line arguments include passing options to a program and passing filenames to a program.

Figure 14.3 copies a file into another file one character at a time. We assume that the executable file for the program is called `mycopy`. A typical command line for the `mycopy` program on a Linux/UNIX system is

```
$ mycopy input output
```

This command line indicates that file `input` is to be copied to file `output`. When the program is executed, if `argc` is not 3 (`mycopy` counts as one of the arguments), the program prints an error message and terminates. Otherwise, array `argv` contains the strings "mycopy", "input" and "output". The second and third arguments on the command line are used as file names by the program. The files are opened using function `fopen`. If both files are opened successfully, characters are read from file `input` and written to file `output` until the end-of-file indicator for file `input` is set. Then the program terminates. The result is an exact copy of file `input`. See the manuals for your system for more information on command-line arguments. [Note: In Visual C++, you can specify the command-line arguments by going to Project Properties > Configuration Properties > Debugging and entering the arguments in the textbox to the right of Command Arguments.]
14.5 Notes on Compiling Multiple-Source-File Programs

It's possible to build programs that consist of multiple source files. There are several considerations when creating programs in multiple files. For example, the definition of a function must be entirely contained in one file—it cannot span two or more files.

In Chapter 5, we introduced the concepts of storage class and scope. We learned that variables declared outside any function definition are of storage class `static` by default and are referred to as global variables. Global variables are accessible to any function defined in the same file after the variable is declared. Global variables also are accessible to functions in other files. However, the global variables must be declared in each file in which they are used. For example, if we define global integer variable `flag` in one file and refer to it in a second file, the second file must contain the declaration

```c
extern int flag;
```
prior to the variable’s use in that file. This declaration uses the storage class specifier `extern` to indicate that variable `flag` is defined either later in the same file or in a different file. The compiler informs the linker that unresolved references to variable `flag` appear in the file (the compiler does not know where `flag` is defined, so it lets the linker attempt to find `flag`). If the linker cannot locate a definition of `flag`, the linker issues an error message and does not produce an executable file. If the linker finds a proper global definition, the linker resolves the references by indicating where `flag` is located.

Software Engineering Observation 14.1

Global variables should be avoided unless application performance is critical because they violate the principle of least privilege.

Just as `extern` declarations can be used to declare global variables to other program files, function prototypes can extend the scope of a function beyond the file in which it’s defined (the `extern` specifier is not required in a function prototype). Simply include the function prototype in each file in which the function is invoked and compile the files together (see Section 13.2). Function prototypes indicate to the compiler that the specified function is defined either later in the same file or in a different file. Again, the compiler does not attempt to resolve references to such a function—that task is left to the linker. If the linker cannot locate a proper function definition, the linker issues an error message.

As an example of using function prototypes to extend the scope of a function, consider any program containing the preprocessor directive `#include <stdio.h>`, which includes in a file the function prototypes for functions such as `printf` and `scanf`. Other functions in the file can use `printf` and `scanf` to accomplish their tasks. The `printf` and `scanf` functions are defined in other files. We do not need to know where they are defined. We’re simply reusing the code in our programs. The linker resolves our references to these functions automatically. This process enables us to use the functions in the standard library.

Software Engineering Observation 14.2

Creating programs in multiple source files facilitates software reusability and good software engineering. Functions may be common to many applications. In such instances, those functions should be stored in their own source files, and each source file should have a corresponding header file containing function prototypes. This enables programmers of different applications to reuse the same code by including the proper header file and compiling their applications with the corresponding source file.

It’s possible to restrict the scope of a global variable or function to the file in which it’s defined. The storage class specifier `static`, when applied to a global variable or a function, prevents it from being used by any function that is not defined in the same file. This is referred to as internal linkage. Global variables and functions that are not preceded by `static` in their definitions have external linkage—they can be accessed in other files if those files contain proper declarations and/or function prototypes.

The global variable declaration

```
static const double PI = 3.14159;
```

creates constant variable `PI` of type `double`, initializes it to `3.14159` and indicates that `PI` is known only to functions in the file in which it’s defined.
The static specifier is commonly used with utility functions that are called only by functions in a particular file. If a function is not required outside a particular file, the principle of least privilege should be enforced by using static. If a function is defined before it's used in a file, static should be applied to the function definition. Otherwise, static should be applied to the function prototype.

When building large programs in multiple source files, compiling the program becomes tedious if small changes are made to one file and the entire program must be recompiled. Many systems provide special utilities that recompile only the modified program file. On Linux/UNIX systems the utility is called make. Utility make reads a file called makefile that contains instructions for compiling and linking the program. Products such as Eclipse™ and Microsoft® Visual C++® provide similar utilities as well. For more information on make utilities, see the manual for your development tool.

### 14.6 Program Termination with exit and atexit

The general utilities library (<stdlib.h>) provides methods of terminating program execution by means other than a conventional return from function main. Function exit forces a program to terminate as if it executed normally. The function often is used to terminate a program when an input error is detected, or if a file to be processed by the program cannot be opened. Function atexit registers a function that should be called upon successful termination of the program—i.e., either when the program terminates by reaching the end of main, or when exit is invoked.

Function atexit takes as an argument a pointer to a function (i.e., the function name). Functions called at program termination cannot have arguments and cannot return a value. Up to 32 functions may be registered for execution at program termination.

Function exit takes one argument. The argument is normally the symbolic constant EXIT_SUCCESS or the symbolic constant EXIT_FAILURE. If exit is called with EXIT_SUCCESS, the implementation-defined value for successful termination is returned to the calling environment. If exit is called with EXIT_FAILURE, the implementation-defined value for unsuccessful termination is returned. When function exit is invoked, any functions previously registered with atexit are invoked in the reverse order of their registration, all streams associated with the program are flushed and closed, and control returns to the host environment.

Figure 14.4 tests functions exit and atexit. The program prompts the user to determine whether the program should be terminated with exit or by reaching the end of main. Function print is executed at program termination in each case.

```c
/* Fig. 14.4: fig14_04.c
Using the exit and atexit functions */
#include <stdio.h>
#include <stdlib.h>

void print( void ); /* prototype */
```

Fig. 14.4 | exit and atexit functions. (Part 1 of 2.)
In Chapters 6–7, we introduced the `const` type qualifier. C also provides the `volatile` type qualifier to suppress various kinds of optimizations. The C standard indicates that when `volatile` is used to qualify a type, the nature of the access to an object of that type is implementation dependent. This usually implies that the variable may be changed by another program or by the computer’s hardware.

```c
int main( void )
{
    int answer; /* user's menu choice */
    atexit( print ); /* register function print */
    printf( "Enter 1 to terminate program with function exit
            \nEnter 2 to terminate program normally\n" );
    scanf( "%d", &answer );

    /* call exit if answer is 1 */
    if ( answer == 1 ) {
        printf( "\nTerminating program with function exit\n" );
        exit( EXIT_SUCCESS );
    } /* end if */

    printf( "\nTerminating program by reaching the end of main\n" );
    return 0; /* indicates successful termination */
} /* end main */

/* display message before termination */
void print( void )
{
    printf( "Executing function print at program termination\nProgram terminated\n" );
} /* end function print */
```

**Fig. 14.4** | exit and atexit functions. (Part 2 of 2.)

### 14.7 volatile Type Qualifier

In Chapters 6–7, we introduced the `const` type qualifier. C also provides the `volatile` type qualifier to suppress various kinds of optimizations. The C standard indicates that when `volatile` is used to qualify a type, the nature of the access to an object of that type is implementation dependent. This usually implies that the variable may be changed by another program or by the computer’s hardware.
14.8 Suffixes for Integer and Floating-Point Constants

C provides integer and floating-point suffixes for specifying the types of integer and floating-point constants. The integer suffixes are: \texttt{u} or \texttt{U} for an \texttt{unsigned integer}, \texttt{l} or \texttt{L} for a \texttt{long integer}, and \texttt{ul}, \texttt{lu}, \texttt{UL} or \texttt{LU} for an unsigned \texttt{long} integer. The following constants are of type \texttt{unsigned}, \texttt{long} and \texttt{unsigned long}, respectively:

\begin{verbatim}
174u 
8358L 
28373ul
\end{verbatim}

If an integer constant is not suffixed, its type is determined by the first type capable of storing a value of that size (first \texttt{int}, then \texttt{long int}, then \texttt{unsigned long int}).

The floating-point suffixes are: \texttt{f} or \texttt{F} for a \texttt{float}, and \texttt{l} or \texttt{L} for a \texttt{long double}. The following constants are of type \texttt{float} and \texttt{long double}, respectively:

\begin{verbatim}
1.28f 
3.14159L
\end{verbatim}

A floating-point constant that is not suffixed is automatically of type \texttt{double}.

14.9 More on Files

Chapter 11 introduced capabilities for processing text files with sequential access and random access. C also provides capabilities for processing binary files, but some computer systems do not support binary files. If binary files are not supported and a file is opened in a binary file mode (Fig. 14.5), the file will be processed as a text file. Binary files should be used instead of text files only in situations where rigid speed, storage and/or compatibility conditions demand binary files. Otherwise, text files are always preferred for their inherent portability and for the ability to use other standard tools to examine and manipulate the file data.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{rb}</td>
<td>Open an existing binary file for reading.</td>
</tr>
<tr>
<td>\texttt{wb}</td>
<td>Create a binary file for writing. If the file already exists, discard the current contents.</td>
</tr>
<tr>
<td>\texttt{ab}</td>
<td>Append; open or create a binary file for writing at end-of-file.</td>
</tr>
<tr>
<td>\texttt{rb+}</td>
<td>Open an existing binary file for update (reading and writing).</td>
</tr>
<tr>
<td>\texttt{wb+}</td>
<td>Create a binary file for update. If the file already exists, discard the current contents.</td>
</tr>
<tr>
<td>\texttt{ab+}</td>
<td>Append; open or create a binary file for update; all writing is done at the end of the file.</td>
</tr>
</tbody>
</table>

\textbf{Performance Tip 14.1}

Use binary files instead of text files in applications that demand high performance.

\textbf{Portability Tip 14.1}

Use text files when writing portable programs.
The standard library also provides function `tmpfile` that opens a temporary file in mode "wb+". Although this is a binary file mode, some systems process temporary files as text files. A temporary file exists until it's closed with `fclose`, or until the program terminates. Microsoft has deprecated this function “for security reasons.”

Figure 14.6 changes the tabs in a file to spaces. The program prompts the user to enter the name of a file to be modified. If the file entered by the user and the temporary file are opened successfully, the program reads characters from the file to be modified and writes them to the temporary file. If the character read is a tab ("\t"), it’s replaced by a space and written to the temporary file. When the end of the file being modified is reached, the file pointers for each file are repositioned to the start of each file with `rewind`. Next, the temporary file is copied into the original file one character at a time. The program prints the original file as it copies characters into the temporary file and prints the new file as it copies characters from the temporary file to the original file to confirm the characters being written.

```c
/* Fig. 14.6: fig14_06.c
Using temporary files */
#include <stdio.h>

int main( void )
{
    FILE *filePtr; /* pointer to file being modified */
    FILE *tempFilePtr; /* temporary file pointer */
    int c; /* define c to hold characters read from a file */
    char fileName[30]; /* create char array */

    printf( "This program changes tabs to spaces.\n"
            "Enter a file to be modified: ");
    scanf( "%29s", fileName );

    /* fopen opens the file */
    if (( filePtr = fopen( fileName, "r+" ) ) != NULL ) {
        /* create temporary file */
        if (( tempFilePtr = tmpfile() ) != NULL ) {
            printf( "\nThe file before modification is:\n" );

            /* read characters from file and place in temporary file */
            while ( ( c = getc( filePtr ) ) != EOF ) {
                putchar( c );
                putc( c == '\t' ? ' ' : c, tempFilePtr );
            } /* end while */

            rewind( tempFilePtr );
            rewind( filePtr );
            printf( "\n\nThe file after modification is:\n" );

            /* read from temporary file and write into original file */
            while ( ( c = getc( tempFilePtr ) ) != EOF ) {
                putchar( c );
                putc( c, filePtr );
            } /* end while */
        } /* end if */
    } /* end if */
```
14.10 Signal Handling

An external asynchronous event, or signal, can cause a program to terminate prematurely. Some events include interrupts (typing \(<\text{Ctrl}\>\text{c}\) on a Linux/UNIX or Windows system), illegal instructions, segmentation violations, termination orders from the operating system and floating-point exceptions (division by zero or multiplying large floating-point values). The signal handling library \(<\text{signal.h}\>\) provides the capability to trap unexpected events with function signal. Function signal receives two arguments—an integer signal number and a pointer to the signal handling function. Signals can be generated by function raise which takes an integer signal number as an argument. Figure 14.7 summarizes the standard signals defined in header file \(<\text{signal.h}\>\).

<table>
<thead>
<tr>
<th>Signal</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>Abnormal termination of the program (such as a call to function abort).</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>An erroneous arithmetic operation, such as a divide by zero or an operation resulting in overflow.</td>
</tr>
<tr>
<td>SIGILL</td>
<td>Detection of an illegal instruction.</td>
</tr>
<tr>
<td>SIGINT</td>
<td>Receipt of an interactive attention signal.</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>An invalid access to storage.</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>A termination request set to the program.</td>
</tr>
</tbody>
</table>

Fig. 14.7 | signal.h standard signals.
Figure 14.8 uses function `signal` to trap an interactive signal (SIGINT). Line 15 calls `signal` with SIGINT and a pointer to function `signalHandler` (remember that the name of a function is a pointer to the beginning of the function). When a signal of type SIGINT occurs, control passes to function `signalHandler`, which prints a message and gives the user the option to continue normal execution of the program. If the user wishes to continue execution, the signal handler is reinitialized by calling `signal` again and control returns to the point in the program at which the signal was detected. In this program, function `raise` (line 24) is used to simulate an interactive signal. A random number between 1 and 50 is chosen. If the number is 25, `raise` is called to generate the signal. Normally, interactive signals are initiated outside the program. For example, typing `<Ctrl> c` during program execution on a Linux/UNIX or Windows system generates an interactive signal that terminates program execution. Signal handling can be used to trap the interactive signal and prevent the program from being terminated.

```c
/* Fig. 14.8: fig14_08.c */
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <time.h>

/* output numbers 1 to 100 */
for (i = 1; i <= 100; i++) {
  x = 1 + rand() % 50; /* generate random number to raise SIGINT */
  if (x == 25) {
    raise(SIGINT);
  } /* end if */
  printf("%4d", i);

  /* output \n when i is a multiple of 10 */
  if (i % 10 == 0) {
    printf("\n");
  } /* end if */
} /* end for */
return 0; /* indicates successful termination */
}
```

Fig. 14.8 | Signal handling. (Part 1 of 2.)
Chapter 14  Other C Topics

14.11 Dynamic Memory Allocation: Functions calloc and realloc

Chapter 12 introduced the notion of dynamically allocating memory using function malloc. As we stated in Chapter 12, arrays are better than linked lists for rapid sorting, searching and data access. However, arrays are normally static data structures. The general
utilities library (`stdlib.h`) provides two other functions for dynamic memory allocation—`calloc` and `realloc`. These functions can be used to create and modify dynamic arrays. As shown in Chapter 7, a pointer to an array can be subscripted like an array. Thus, a pointer to a contiguous portion of memory created by `calloc` can be manipulated as an array. Function `calloc` dynamically allocates memory for an array. The prototype for `calloc` is

```c
void *calloc( size_t nmemb, size_t size );
```

Its two arguments represent the number of elements (`nmemb`) and the size of each element (`size`). Function `calloc` also initializes the elements of the array to zero. The function returns a pointer to the allocated memory, or a `NULL` pointer if the memory is not allocated. The primary difference between `malloc` and `calloc` is that `calloc` clears the memory it allocates and `malloc` does not.

Function `realloc` changes the size of an object allocated by a previous call to `malloc`, `calloc` or `realloc`. The original object’s contents are not modified provided that the amount of memory allocated is larger than the amount allocated previously. Otherwise, the contents are unchanged up to the size of the new object. The prototype for `realloc` is

```c
void *realloc( void *ptr, size_t size );
```

The two arguments are a pointer to the original object (`ptr`) and the new size of the object (`size`). If `ptr` is `NULL`, `realloc` works identically to `malloc`. If `size` is 0 and `ptr` is not `NULL`, the memory for the object is freed. Otherwise, if `ptr` is not `NULL` and `size` is greater than zero, `realloc` tries to allocate a new block of memory for the object. If the new space cannot be allocated, the object pointed to by `ptr` is unchanged. Function `realloc` returns either a pointer to the reallocated memory, or a `NULL` pointer to indicate that the memory was not reallocated.

### 14.12 Unconditional Branching with `goto`

Throughout the text we have stressed the importance of using structured programming techniques to build reliable software that is easy to debug, maintain and modify. In some cases, performance is more important than strict adherence to structured programming techniques. In these cases, some unstructured programming techniques may be used. For example, we can use `break` to terminate execution of a repetition structure before the loop continuation condition becomes false. This saves unnecessary repetitions of the loop if the task is completed before loop termination.

Another instance of unstructured programming is the `goto statement`—an unconditional branch. The result of the `goto` statement is a change in the flow of control of the program to the first statement after the `label` specified in the `goto` statement. A label is an identifier followed by a colon. A label must appear in the same function as the `goto` statement that refers to it. Figure 14.9 uses `goto` statements to loop ten times and print the counter value each time. After initializing `count` to 1, line 11 tests `count` to determine whether it’s greater than 10 (the label `start` is skipped because labels do not perform any action). If so, control is transferred from the `goto` to the first statement after the label `end` (which appears at line 20). Otherwise, lines 15–16 print and increment `count`, and control transfers from the `goto` (line 18) to the first statement after the label `start` (which appears at line 9).
In Chapter 3, we stated that only three control structures are required to write any program—sequence, selection and repetition. When the rules of structured programming are followed, it’s possible to create deeply nested control structures from which it’s difficult to efficiently escape. Some programmers use `goto` statements in such situations as a quick exit from a deeply nested structure. This eliminates the need to test multiple conditions to escape from a control structure.

```c
/* Fig. 14.9: fig14_09.c
  Using goto */
#include <stdio.h>

int main( void )
{
  int count = 1; /* initialize count */

  start: /* label */
    if ( count > 10 ) {
      goto end;
    } /* end if */

    printf("%d ", count);
    count++;
  goto start; /* goto start on line 9 */

  end: /* label */
    putchar( '\n' );

  return 0; /* indicates successful termination */
} /* end main */
```

**Performance Tip 14.2**
The `goto` statement can be used to exit deeply nested control structures efficiently.

**Software Engineering Observation 14.3**
The `goto` statement should be used only in performance-oriented applications. The `goto` statement is unstructured and can lead to programs that are more difficult to debug, maintain and modify.

**Summary**

**Section 14.2 Redirecting I/O**
- On many computer systems it’s possible to redirect input to a program and output from a program.
- Input is redirected from the command line using the redirect input symbol `<` or using a pipe `|`.
• Output is redirected from the command line using the redirect output symbol (>) or the append output symbol (>>). The redirect output symbol simply stores the program output in a file, and the append output symbol appends the output to the end of a file.

Section 14.3 Variable-Length Argument Lists
• The macros and definitions of the variable arguments header <stdarg.h> provide the capabilities necessary to build functions with variable-length argument lists.
• An ellipsis (…) in a function prototype indicates a variable number of arguments.
• Type va_list is suitable for holding information needed by macros va_start, va_arg and va_end. To access the arguments in a variable-length argument list, an object of type va_list must be declared.
• Invoke macro va_start before accessing the arguments of a variable-length argument list. The macro initializes the object declared with va_list for use by the va_arg and va_end macros.
• Macro va_arg expands to an expression of the value and type of the next argument in the variable length argument list. Each invocation of va_arg modifies the object declared with va_list so that the object points to the next argument in the list.
• Macro va_end facilitates a normal return from a function whose variable argument list was referred to by the va_start macro.

Section 14.4 Using Command-Line Arguments
• On many systems it’s possible to pass arguments to main from the command line by including the parameters int argc and char *argv[] in the parameter list of main. Parameter argc receives the number of command-line arguments. Parameter argv is an array of strings in which the actual command-line arguments are stored.

Section 14.5 Notes on Compiling Multiple-Source-File Programs
• A function definition must be entirely contained in one file—it cannot span two or more files.
• Global variables must be declared in each file in which they are used.
• Function prototypes can extend the scope of a function beyond the file in which it’s defined. This is accomplished by including the function prototype in each file in which the function is invoked and compiling the files together.
• The storage class specifier static, when applied to a global variable or a function, prevents it from being used by any function that is not defined in the same file. This is referred to as internal linkage. Global variables and functions that are not preceded by static in their definitions have external linkage—they can be accessed in other files if those files contain proper declarations or function prototypes.
• The static specifier is commonly used with utility functions that are called only by functions in a particular file. If a function is not required outside a particular file, the principle of least privilege should be enforced by using static.
• When building large programs in multiple source files, compiling the program becomes tedious if small changes are made to one file and the entire program must be recompiled. Many systems provide special utilities that recompile only the modified program file. On Linux/UNIX systems the utility is called make. Utility make reads a file called makefile that contains instructions for compiling and linking the program.

Section 14.6 Program Termination with exit and atexit
• Function exit forces a program to terminate as if it executed normally.
• Function atexit registers a function to be called upon normal termination of the program—i.e., either when the program terminates by reaching the end of main or when exit is invoked.
• Function atexit takes a pointer to a function as an argument. Functions called at program termination cannot have arguments and cannot return a value. Up to 32 functions may be registered for execution at program termination.

• Function exit takes one argument. The argument is normally the symbolic constant EXIT_SUCCESS or the symbolic constant EXIT_FAILURE. If exit is called with EXIT_SUCCESS, the implementation-defined value for successful termination is returned to the calling environment. If exit is called with EXIT_FAILURE, the implementation-defined value for unsuccessful termination is returned.

• When function exit is invoked, any functions registered with atexit are invoked in the reverse order of their registration, all streams associated with the program are flushed and closed, and control returns to the host environment.

Section 14.7 volatile Type Qualifier

• The C standard indicates that when volatile is used to qualify a type, the nature of the access to an object of that type is implementation dependent.

Section 14.8 Suffixes for Integer and Floating-Point Constants

• C provides integer and floating-point suffixes for specifying the types of integer and floating-point constants. The integer suffixes are: u or U for an unsigned integer, l or L for a long integer, and ul or UL for an unsigned long integer. If an integer constant is not suffixed, its type is determined by the first type capable of storing a value of that size (first int, then long int, then unsigned long int). The floating-point suffixes are: f or F for a float, and l or L for a long double. A floating-point constant that is not suffixed is of type double.

Section 14.9 More on Files

• C provides capabilities for processing binary files, but some computer systems do not support binary files. If binary files are not supported and a file is opened in a binary file mode, the file will be processed as a text file.

• Function tmpfile opens a temporary file in mode "wb+". Although this is a binary file mode, some systems process temporary files as text files. A temporary file exists until it’s closed with fclose or until the program terminates.

Section 14.10 Signal Handling

• The signal handling library enables trapping of unexpected events with function signal. Function signal receives two arguments—an integer signal number and a pointer to the signal-handling function.

• Signals can also be generated with function raise and an integer argument.

Section 14.11 Dynamic Memory Allocation: Functions calloc and realloc

• The general utilities library (<stdlib.h>) provides two functions for dynamic memory allocation—calloc and realloc. These functions can be used to create dynamic arrays.

• Function calloc receives two arguments—the number of elements (nmemb) and the size of each element (size)—and initializes the elements of the array to zero. The function returns either a pointer to the allocated memory, or a NULL pointer if the memory is not allocated.

• Function realloc changes the size of an object allocated by a previous call to malloc, calloc or realloc. The original object’s contents are not modified provided that the amount of memory allocated is larger than the amount allocated previously.

• Function realloc takes two arguments—a pointer to the original object (ptr) and the new size of the object (size). If ptr is NULL, realloc works identically to malloc. If size is 0 and the pointer received is not NULL, the memory for the object is freed. Otherwise, if ptr is not NULL and size
is greater than zero, realloc tries to allocate a new block of memory for the object. If the new space cannot be allocated, the object pointed to by \( \text{ptr} \) is unchanged. Function realloc returns either a pointer to the reallocated memory, or a NULL pointer.

**Section 14.12 Unconditional Branching with goto**

- The result of the goto statement is a change in the flow of control of the program. Program execution continues at the first statement after the label specified in the goto statement.
- A label is an identifier followed by a colon. A label must appear in the same function as the goto statement that refers to it.

**Terminology**

append output symbol >> 509
argc 511
argv 511
atexit 514
calloc 521
const type qualifier 515
dynamic array 521
ellipsis (…) in a function prototype 509
event 518
exit function 514
EXIT_FAILURE 514
EXIT_SUCCESS 514
extern 513
external linkage 513
f or F for a float 516
float 516
floating-point exception 518
goto statement 521
illegal instruction 518
internal linkage 513
interrupt 518
l or L for a long double 516
label 521
long double 516
long int 516
long integer 516
make 514
makefile 514
pipe symbol (|) 508
piping 508
raise 518
redirect input from a file 508
redirect input symbol < 508
redirect output symbol > 509
segmentation violation 518
signal 518
signal handling library 518
<stdio.h> 518
static keyword 512
static data structure 520
<stdio.h> header file 521
temporary file 517
tmppfile 517
trap 518
unsigned integer 516
unsigned long int 516
va_arg 511
va_end 511
va_list 511
va_start 511
variable arguments header stdarg.h 509
variable-length argument list 509
volatile type qualifier 515

**Self-Review Exercise**

14.1 Fill in the blanks in each of the following:

a) The ________ symbol redirects input data from a file rather than the keyboard.

b) The ________ symbol is used to redirect the screen output so that it’s placed in a file.

c) The ________ symbol is used to append the output of a program to the end of a file.

d) A(n) ________ directs the output of one program to be the input of another program.

e) A(n) ________ in the parameter list of a function indicates that the function can receive a variable number of arguments.

f) Macro ________ must be invoked before the arguments in a variable-length argument list can be accessed.

g) Macro ________ accesses the individual arguments of a variable-length argument list.
h) Macro ______ facilitates a normal return from a function whose variable argument list was referred to by macro va_start.

i) Argument ______ of main receives the number of arguments in a command line.

j) Argument ______ of main stores command-line arguments as character strings.

k) Linux/UNIX utility ______ reads a file called ______ that contains instructions for compiling and linking a program consisting of multiple source files.

l) Function ______ forces a program to terminate execution.

m) Function ______ registers a function to be called upon normal program termination.

n) An integer or floating-point ______ can be appended to an integer or floating-point constant to specify the exact type of the constant.

o) Function ______ opens a temporary file that exists until it’s closed or program execution terminates.

p) Function ______ can be used to trap unexpected events.

q) Function ______ generates a signal from within a program.

r) Function ______ dynamically allocates memory for an array and initializes the elements to zero.

s) Function ______ changes the size of a block of previously allocated dynamic memory.

Answers to Self-Review Exercise

14.1 a) redirect input (<). b) redirect output (>). c) append output (>>). d) pipe (|). e) ellipsis (...). f) va_start. g) va_arg. h) va_end. i) argc. j) argv. k) make, makefile. l) exit. m) atexit. n) suffix. o) tmpfile. p) signal. q) raise. r) calloc. s) realloc.

Exercises

14.2 (Variable-Length Argument List: Calculating Products) Write a program that calculates the product of a series of integers that are passed to function product using a variable-length argument list. Test your function with several calls, each with a different number of arguments.

14.3 (Printing Command-Line Arguments) Write a program that prints the command-line arguments of the program.

14.4 (Sorting Integers) Write a program that sorts an array of integers into ascending order or descending order. Use command-line arguments to pass either argument -a for ascending order or -d for descending order. [Note: This is the standard format for passing options to a program in UNIX.]

14.5 (Temporary Files) Write a program that places a space between each character in a file. The program should first write the contents of the file being modified into a temporary file with spaces between each character, then copy the file back to the original file. This operation should overwrite the original contents of the file.

14.6 (Signal Handling) Read the manuals for your compiler to determine what signals are supported by the signal handling library (<signal.h>). Write a program that contains signal handlers for the standard signals SIGABRT and SIGINT. The program should test the trapping of these signals by calling function abort to generate a signal of type SIGABRT and by typing <Ctrl> c to generate a signal of type SIGINT.

14.7 (Dynamic Array Allocation) Write a program that dynamically allocates an array of integers. The size of the array should be input from the keyboard. The elements of the array should be assigned values input from the keyboard. Print the values of the array. Next, reallocate the memory for the array to 1/2 of the current number of elements. Print the values remaining in the array to confirm that they match the first half of the values in the original array.
14.8  *(Command-Line Arguments)* Write a program that takes two command-line arguments that are file names, reads the characters from the first file one at a time and writes the characters in reverse order to the second file.

14.9  *(goto Statement)* Write a program that uses goto statements to simulate a nested looping structure that prints a square of asterisks as follows:

```
*****
*   *
*   *
*   *
*****
```

The program should use only the following three printf statements:

```c
printf("\n");
```
A wise skepticism is the first attribute of a good critic.
—James Russell Lowell

... no science of behavior can change the essential nature of man ...
—Burrhus Frederic Skinner

Nothing can have value without being an object of utility.
—Karl Marx

Knowledge is the conformity of the object and the intellect.
—Averroës

Many things, having full reference To one consent, may work contrariously ...
—William Shakespeare

Objectives
In this chapter you’ll learn:

■ C++ enhancements to C.
■ The header files of the C++ Standard Library.
■ To use inline functions.
■ To use references.
■ To use default arguments.
■ To use the unary scope resolution operator to access a global variable.
■ To overload functions.
■ To create and use function templates that perform identical operations different types.
15.1 Introduction

We now begin the second section of this unique text. The first 14 chapters presented a thorough treatment of procedural programming and top-down program design with C. The C++ section (Chapters 15–24) introduces two additional programming paradigms—object-oriented programming (with classes, encapsulation, objects, operator overloading, inheritance and polymorphism) and generic programming (with function templates and class templates). These chapters emphasize “crafting valuable classes” to create reusable software componentry.

15.2 C++

C++ improves on many of C’s features and provides object-oriented-programming (OOP) capabilities that increase software productivity, quality and reusability. This chapter discusses many of C++’s enhancements to C.

C’s designers and early implementers never anticipated that the language would become such a phenomenon. When a programming language becomes as entrenched as C, new requirements demand that the language evolve rather than simply be displaced by a new language. C++ was developed by Bjarne Stroustrup at Bell Laboratories and was originally called “C with classes.” The name C++ includes C’s increment operator (++) to indicate that C++ is an enhanced version of C.

Chapters 15–24 provide an introduction to the version of C++ standardized in the United States through the American National Standards Institute (ANSI) and worldwide through the International Standards Organization (ISO). We have done a careful walkthrough of the ANSI/ISO C++ standard document and audited our presentation against it for completeness and accuracy. However, C++ is a rich language, and there are some subtleties in the language and some advanced subjects that we have not covered. If you need additional technical details on C++, we suggest that you read the C++ standard document, which you can purchase from the ANSI website

[webstore.ansi.org/ansidocstore/product.asp?sku=INCITS%2FISO%2FIEC+14882%2D2003]

The title of the document is “Programming languages—C++” and its document number is INCITS/ISO/IEC 14882-2003.
15.3 A Simple Program: Adding Two Integers

This section revisits the addition program of Fig. 2.8 and illustrates several important features of the C++ language as well as some differences between C and C++. C file names have the .c (lowercase) extension. C++ file names can have one of several extensions, such as .cpp, .cxx or .C (uppercase). We use the extension .cpp.

Figure 15.1 uses C++-style input and output to obtain two integers typed by a user at the keyboard, computes the sum of these values and outputs the result. Lines 1 and 2 each begin with //, indicating that the remainder of each line is a comment. C++ allows you to begin a comment with // and use the remainder of the line as comment text. A // comment is a maximum of one line long. C++ programmers may also use /*...*/ C-style comments, which can be more than one line long.

```cpp
#include <iostream> // allows program to perform input and output

int main()
{
    int number1; // first integer to add
    std::cout << "Enter first integer: "; // prompt user for data
    std::cin >> number1; // read first integer from user into number1

    int number2; // second integer to add
    int sum; // sum of number1 and number2
    std::cout << "Enter second integer: "; // prompt user for data
    std::cin >> number2; // read second integer from user into number2
    sum = number1 + number2; // add the numbers; store result in sum
    std::cout << "Sum is " << sum << std::endl; // display sum; end line
}
```

Enter first integer: 45
Enter second integer: 72
Sum is 117

Fig. 15.1 | Addition program that displays the sum of two numbers.

The C++ preprocessor directive in line 3 exhibits the standard C++ style for including header files from the standard library. This line tells the C++ preprocessor to include the contents of the input/output stream header file <iostream>. This file must be included for any program that outputs data to the screen or inputs data from the keyboard using C++-style stream input/output. We discuss iostream’s many features in detail in Chapter 23, Stream Input/Output.

As in C, every C++ program begins execution with function main (line 5). Keyword int to the left of main indicates that main returns an integer value. C++ requires you to specify the return type, possibly void, for all functions. In C++, specifying a parameter list with empty parentheses is equivalent to specifying a void parameter list in C. In C, using empty parentheses in a function definition or prototype is dangerous. It disables compile-
time argument checking in function calls, which allows the caller to pass any arguments to the function. This could lead to runtime errors.

Common Programming Error 15.1

Omitting the return type in a C++ function definition is a syntax error.

Line 7 is a familiar variable declaration. Declarations can be placed almost anywhere in a C++ program, but they must appear before their corresponding variables are used in the program. For example, in Fig. 15.1, the declaration in line 7 could have been placed immediately before line 10, the declaration in line 12 could have been placed immediately before line 16 and the declaration in line 13 could have been placed immediately before line 17.

Good Programming Practice 15.1

Always place a blank line between a declaration and adjacent executable statements. This makes the declarations stand out in the program, enhancing program clarity.

Line 9 uses the standard output stream object—\texttt{std::cout}—and the stream insertion operator, \texttt{<<}, to display the string "Enter first integer: ". Output and input in C++ are accomplished with streams of characters. Thus, when line 9 executes, it sends the stream of characters "Enter first integer: " to \texttt{std::cout}, which is normally “connected” to the screen. We like to pronounce the preceding statement as “\texttt{std::cout gets} the character string "Enter first integer: ".”

Line 10 uses the standard input stream object—\texttt{std::cin}—and the stream extraction operator, \texttt{>>}, to obtain a value from the keyboard. Using the stream extraction operator with \texttt{std::cin} takes character input from the standard input stream, which is usually the keyboard. We like to pronounce the preceding statement as, “\texttt{std::cin gives a value to number1}” or simply “\texttt{std::cin gives number1}.”

When the computer executes the statement in line 10, it waits for the user to enter a value for variable \texttt{number1}. The user responds by typing an integer (as characters), then pressing the \texttt{Enter} key. The computer converts the character representation of the number to an integer and assigns this value to the variable \texttt{number1}.

Line 15 displays "Enter second integer: " on the screen, prompting the user to take action. Line 16 obtains a value for variable \texttt{number2} from the user.

The assignment statement in line 17 calculates the sum of the variables \texttt{number1} and \texttt{number2} and assigns the result to variable \texttt{sum}. Line 18 displays the character string \texttt{Sum is} followed by the numerical value of variable \texttt{sum} followed by \texttt{std::endl}—a so-called stream manipulator. The name \texttt{endl} is an abbreviation for “end line.” The \texttt{std::endl} stream manipulator outputs a newline, then “flushes the output buffer.” This simply means that, on some systems where outputs accumulate in the machine until there are enough to “make it worthwhile” to display on the screen, \texttt{std::endl} forces any accumulated outputs to be displayed at that moment. This can be important when the outputs are prompting the user for an action, such as entering data.

We place \texttt{std::} before \texttt{cout}, \texttt{cin} and \texttt{endl}. This is required when we use standard C++ header files. The notation \texttt{std::cout} specifies that we’re using a name, in this case \texttt{cout}, that belongs to “namespace” \texttt{std}. Namespaces are an advanced C++ feature that we do not discuss in these introductory C++ chapters. For now, you should simply remember to include \texttt{std::} before each mention of \texttt{cout}, \texttt{cin} and \texttt{endl} in a program. This can be
cumbersome—in Fig. 15.3, we introduce the using statement, which will enable us to avoid placing std:: before each use of a namespace std name.

The statement in line 18 outputs values of different types. The stream insertion operator “knows” how to output each type of data. Using multiple stream insertion operators (<<) in a single statement is referred to as concatenating, chaining or cascading stream insertion operations.

Calculations can also be performed in output statements. We could have combined the statements in lines 17 and 18 into the statement

```cpp
std::cout << "Sum is " << number1 + number2 << std::endl;
```

thus eliminating the need for the variable sum.

You’ll notice that we did not have a return 0; statement at the end of main in this example. According to the C++ standard, if program execution reaches the end of main without encountering a return statement, it’s assumed that the program terminated successfully—exactly as when the last statement in main is a return statement with the value 0. For that reason, we omit the return statement at the end of main in our C++ programs.

A powerful C++ feature is that users can create their own types called classes (we introduce this capability in Chapter 16 and explore it in depth in Chapters 17–18). Users can then “teach” C++ how to input and output values of these new data types using the >> and << operators (this is called operator overloading—a topic we explore in Chapter 19).

## 15.4 C++ Standard Library

C++ programs consist of pieces called classes and functions. You can program each piece that you may need to form a C++ program. Instead, most C++ programmers take advantage of the rich collections of existing classes and functions in the C++ Standard Library. Thus, there are really two parts to learning the C++ “world.” The first is learning the C++ language itself; the second is learning how to use the classes and functions in the C++ Standard Library. Throughout the book, we discuss many of these classes and functions. P J. Plauger’s book, *The Standard C Library* (Englewood Cliffs, NJ: Prentice Hall PTR, 1992), is a must read for programmers who need a deep understanding of the Standard C library functions that are included in C++, how to implement them and how to use them to write portable code. The standard class libraries generally are provided by compiler vendors. Many special-purpose class libraries are supplied by independent software vendors.

### Software Engineering Observation 15.1

Use a “building-block” approach to create programs. Avoid reinventing the wheel. Use existing pieces wherever possible. Called software reuse, this practice is central to object-oriented programming.

### Software Engineering Observation 15.2

When programming in C++, you typically will use the following building blocks: classes and functions from the C++ Standard Library, classes and functions you and your colleagues create and classes and functions from various popular third-party libraries.

The advantage of creating your own functions and classes is that you’ll know exactly how they work. You’ll be able to examine the C++ code. The disadvantage is the time-con-
suming and complex effort that goes into designing, developing and maintaining new functions and classes that are correct and that operate efficiently.

Performance Tip 15.1

Using C++ Standard Library functions and classes instead of writing your own versions can improve program performance, because they are written to perform efficiently. This technique also shortens program development time.

Portability Tip 15.1

Using C++ Standard Library functions and classes instead of writing your own improves program portability, because they are included in every C++ implementation.

15.5 Header Files

The C++ Standard Library is divided into many portions, each with its own header file. The header files contain the function prototypes for the related functions that form each portion of the library. The header files also contain definitions of various class types and functions, as well as constants needed by those functions. A header file “instructs” the compiler on how to interface with library and user-written components.

Figure 15.2 lists some common C++ Standard Library header files. Header file names ending in .h are “old-style” header files that have been superceded by the C++ Standard Library header files.

<table>
<thead>
<tr>
<th>C++ Standard Library header file</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;iostream&gt;</code></td>
<td>Contains function prototypes for the C++ standard input and standard output functions. This header file replaces header file <code>&lt;iostream.h&gt;</code>. This header is discussed in detail in Chapter 23, Stream Input/Output.</td>
</tr>
<tr>
<td><code>&lt;iomanip&gt;</code></td>
<td>Contains function prototypes for stream manipulators that format streams of data. This header file replaces header file <code>&lt;iomanip.h&gt;</code>. This header is used in Chapter 23, Stream Input/Output.</td>
</tr>
<tr>
<td><code>&lt;cmath&gt;</code></td>
<td>Contains function prototypes for math library functions. This header file replaces header file <code>&lt;math.h&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;cstdlib&gt;</code></td>
<td>Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. This header file replaces header file <code>&lt;stdlib.h&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;ctime&gt;</code></td>
<td>Contains function prototypes and types for manipulating the time and date. This header file replaces header file <code>&lt;time.h&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;vector&gt;</code>, <code>&lt;list&gt;</code>, <code>&lt;deque&gt;</code>, <code>&lt;queue&gt;</code>, <code>&lt;stack&gt;</code>, <code>&lt;map&gt;</code>, <code>&lt;set&gt;</code>, <code>&lt;bitset&gt;</code></td>
<td>These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program’s execution.</td>
</tr>
</tbody>
</table>
### C++ Standard Library header files

<table>
<thead>
<tr>
<th>Header File</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| `<cstdlib>` | Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. This header file replaces header file `<ctype.h>`..
| `<cstring>` | Contains function prototypes for C-style string-processing functions. This header file replaces header file `<string.h>`.
| `<typeinfo>` | Contains classes for runtime type identification (determining data types at execution time).
| `<exception>`, `<stdexcept>` | These header files contain classes that are used for exception handling (discussed in Chapter 24, Exception Handling).
| `<memory>` | Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 24, Exception Handling.
| `<fstream>` | Contains function prototypes for functions that perform input from files on disk and output to files on disk. This header file replaces header file `<fstream.h>`.
| `<string>` | Contains the definition of class `string` from the C++ Standard Library.
| `<sstream>` | Contains function prototypes for functions that perform input from strings in memory and output to strings in memory.
| `<functional>` | Contains classes and functions used by C++ Standard Library algorithms.
| `<iterator>` | Contains classes for accessing C++ Standard Library container data.
| `<algorithm>` | Contains functions for manipulating container data.
| `<cassert>` | Contains macros for adding diagnostics that aid program debugging. This replaces header file `<assert.h>` from pre-standard C++.
| `<cfloat>` | Contains the floating-point size limits of the system. This header file replaces header file `<float.h>`.
| `<climits>` | Contains the integral size limits of the system. This header file replaces header file `<limits.h>`.
| `<cstdio>` | Contains function prototypes for the C-style standard input/output library functions and information used by them. This header file replaces header file `<stdio.h>`.
| `<locale>` | Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, and so on).
| `<limits>` | Contains classes for defining the numerical data type limits on each computer platform.
| `<utility>` | Contains classes and functions that are used by many C++ Standard Library header files.
You can create custom header files. Programmer-defined header files should end in .h. A programmer-defined header file can be included by using the `#include` preprocessor directive. For example, the header file `square.h` can be included in a program by placing the directive `#include "square.h"` at the beginning of the program.

### 15.6 Inline Functions

Implementing a program as a set of functions is good from a software engineering standpoint, but function calls involve execution-time overhead. C++ provides inline functions to help reduce function call overhead—especially for small functions. Placing the qualifier `inline` before a function’s return type in the function definition “advises” the compiler to generate a copy of the function’s code in place (when appropriate) to avoid a function call. The trade-off is that multiple copies of the function code are inserted in the program (often making the program larger) rather than there being a single copy of the function to which control is passed each time the function is called. The compiler can ignore the `inline` qualifier and typically does so for all but the smallest functions.

**Software Engineering Observation 15.3**

Changea to an inline function could require clients of the function to be recompiled. This can be significant in program development and maintenance situations.

**Performance Tip 15.2**

Using inline functions can reduce execution time but may increase program size.

**Software Engineering Observation 15.4**

The inline qualifier should be used only with small, frequently used functions.

Figure 15.3 uses inline function `cube` (lines 11–14) to calculate the volume of a cube of side length `side`. Keyword `const` in the parameter list of function `cube` tells the compiler that the function does not modify variable `side`. This ensures that the value of `side` is not changed by the function when the calculation is performed. Notice that the complete definition of function `cube` appears before it’s used in the program. This is required so that the compiler knows how to expand a cube function call into its inlined code. For this reason, reusable inline functions are typically placed in header files, so that their definitions can be included in each source file that uses them.

**Software Engineering Observation 15.5**

The `const` qualifier should be used to enforce the principle of least privilege. Using the principle of least privilege to properly design software can greatly reduce debugging time and improper side effects, and can make a program easier to modify and maintain.

```cpp
1 // Fig. 15.3: fig15_03.cpp
2 // Using an inline function to calculate the volume of a cube.
3 #include <iostream>
```

Fig. 15.3 | inline function that calculates the volume of a cube. (Part 1 of 2.)
Lines 4–6 are using statements that help us eliminate the need to repeat the `std::` prefix. Once we include these using statements, we can write `cout` instead of `std::cout`, `cin` instead of `std::cin` and `endl` instead of `std::endl`, in the remainder of the program. From this point forward, each C++ example contains one or more using statements.

In place of lines 4–6, many programmers prefer to use the declaration

```
using namespace std;
```

which enables a program to use all the names in any standard C++ header file (such as `<iostream>`) that a program might include. From this point forward in our C++ programs, we’ll use the preceding declaration in our programs.

The `for` statement’s condition (line 20) evaluates to either 0 (false) or nonzero (true). This is consistent with C. C++ also provides type `bool` for representing boolean (true/false) values. The two possible values of a `bool` are the keywords `true` and `false`. When `true`
and `false` are converted to integers, they become the values 1 and 0, respectively. When non-boolean values are converted to type `bool`, non-zero values become `true`, and zero or null pointer values become `false`. Figure 15.4 lists the keywords common to C and C++ and the keywords unique to C++.

### 15.7 References and Reference Parameters

Two ways to pass arguments to functions in many programming languages are pass-by-value and pass-by-reference. When an argument is passed by value, a copy of the argument's value is made and passed (on the function call stack) to the called function. Changes to the copy do not affect the original variable's value in the caller. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. Each argument that has been passed in the programs in this chapter so far has been passed by value.

---

**Performance Tip 15.3**

One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

**Reference Parameters**

This section introduces reference parameters—the first of two means that C++ provides for performing pass-by-reference. With pass-by-reference, the caller gives the called function the ability to access the caller’s data directly, and to modify that data if the called function chooses to do so.
Pass-by-reference is an alias for its corresponding argument in a function call. To indicate that a function parameter is passed by reference, simply follow the parameter’s type in the function prototype by an ampersand (&); use the same notation when listing the parameter’s type in the function header. For example, the following declaration in a function header

```cpp
int &count
```

when read from right to left is pronounced “count is a reference to an int.” In the function call, simply mention the variable by name to pass it by reference. Then, mentioning the variable by its parameter name in the body of the called function actually refers to the original variable in the calling function, and the original variable can be modified directly by the called function. As always, the function prototype and header must agree.

**Passing Arguments by Value and by Reference**

Figure 15.5 compares pass-by-value and pass-by-reference with reference parameters. The “styles” of the arguments in the calls to function `squareByValue` (line 17) and function `squareByReference` (line 22) are identical—both variables are simply mentioned by name in the function calls. Without checking the function prototypes or function definitions, it’s not possible to tell from the calls alone whether either function can modify its arguments. Because function prototypes are mandatory, however, the compiler has no trouble resolving the ambiguity. Recall that a function prototype tells the compiler the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters, and the order in which they are expected. The compiler uses this information to validate function calls. In C, function prototypes are not required. Making them mandatory in C++ enables **type-safe linkage**, which ensures that the types of the arguments conform to the types of the parameters. Otherwise, the compiler reports an error. Locating such type errors at compile time helps prevent the runtime errors that can occur in C when arguments of incorrect data types are passed to functions.

```cpp
int squareByValue( int ); // function prototype (value pass)
```

---

**Performance Tip 15.4**

Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.

**Software Engineering Observation 15.6**

Pass-by-reference can weaken security; the called function can corrupt the caller’s data.

Later, we’ll show how to achieve the performance advantage of pass-by-reference while simultaneously achieving the software engineering advantage of protecting the caller’s data from corruption.

Fig. 15.5 | Passing arguments by value and by reference. (Part 1 of 2.)
15.7 References and Reference Parameters

```cpp
void squareByReference(int &); // function prototype (reference pass)

int main()
{
    int x = 2; // value to square using squareByValue
    int z = 4; // value to square using squareByReference

    // demonstrate squareByValue
    cout << "x = " << x << " before squareByValue\n"
    << squareByValue( x ) << endl;
    cout << "Value returned by squareByValue: "
    << x << " after squareByValue\n" << endl;

    // demonstrate squareByReference
    cout << "z = " << z << " before squareByReference" << endl;
    squareByReference( z );
    cout << "z = " << z << " after squareByReference" << endl;
} // end main

// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue(int number)
{
    return number *= number; // caller's argument not modified
} // end function squareByValue

// squareByReference multiplies numberRef by itself and stores the result
// in the variable to which numberRef refers in the caller
void squareByReference(int &numberRef)
{
    numberRef *= numberRef; // caller's argument modified
} // end function squareByReference
```

Fig. 15.5 Passing arguments by value and by reference. (Part 2 of 2.)

**Common Programming Error 15.2**
Because reference parameters are mentioned only by name in the body of the called function, you might inadvertently treat reference parameters as pass-by-value parameters. This can cause unexpected side effects if the original copies of the variables are changed by the function.

**Performance Tip 15.5** For passing large objects efficiently, use a constant reference parameter to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object. The called function will not be able to modify the object in the caller.
To specify a reference to a constant, place the const qualifier before the type specifier in the parameter declaration. Note in line 35 of Fig. 15.5 the placement of & in the parameter list of function squareByReference. Some C++ programmers prefer to write int& numberRef with the ampersand abutting int—both forms are equivalent to the compiler.

**References as Aliases within a Function**

References can also be used as aliases for other variables within a function (although they typically are used with functions as shown in Fig. 15.5). For example, the code

```cpp
int count = 1; // declare integer variable count
int &cRef = count; // create cRef as an alias for count
cRef++; // increment count (using its alias cRef)
```

increments variable count by using its alias cRef. Reference variables must be initialized in their declarations, as we show in line 9 of both Fig. 15.6 and Fig. 15.7, and cannot be reassigned as aliases to other variables. Once a reference is declared as an alias for a variable, all operations “performed” on the alias (i.e., the reference) are actually performed on the original variable. The alias is simply another name for the original variable. Taking the address of a reference and comparing references do not cause syntax errors; rather, each operation occurs on the variable for which the reference is an alias. Unless it’s a reference to a constant, a reference argument must be an lvalue (e.g., a variable name), not a constant or expression that returns an rvalue (e.g., the result of a calculation).

```cpp
1 // Fig. 15.6: fig15_06.cpp
2 // References must be initialized.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int x = 3;
9     int &y = x; // y refers to (is an alias for) x
10    cout << "x = " << x << endl << "y = " << y << endl;
11    y = 7; // actually modifies x
12    cout << "x = " << x << endl << "y = " << y << endl;
13 } // end main
```

**Software Engineering Observation 15.7**

Many programmers do not declare parameters passed by value as const, even when the called function should not modify the passed argument. Keyword const in this context would protect only a copy of the original argument, not the original argument itself, which when passed by value is safe from modification by the called function.

**Software Engineering Observation 15.8**

For the combined reasons of clarity and performance, many C++ programmers prefer that modifiable arguments be passed to functions by using pointers, small nonmodifiable arguments be passed by value and large nonmodifiable arguments be passed by using references to constants.
Returning a Reference from a Function

Returning references from functions can be dangerous. When returning a reference to a variable declared in the called function, the variable should be declared static within that function. Otherwise, the reference refers to an automatic variable that is discarded when the function terminates; such a variable is "undefined," and the program's behavior is unpredictable. References to undefined variables are called dangling references.

**Common Programming Error 15.3**

Not initializing a reference variable when it's declared is a compilation error, unless the declaration is part of a function's parameter list. Reference parameters are initialized when the function in which they're declared is called.

**Common Programming Error 15.4**

Attempting to reassign a previously declared reference to be an alias to another variable is a logic error. The value of the other variable is simply assigned to the variable for which the reference is already an alias.
Chapter 15  C++ as a Better C; Introducing Object Technology

**Common Programming Error 15.5**

Returning a reference to an automatic variable in a called function is a logic error. Some compilers issue a warning when this occurs.

**Error Messages for Uninitialized References**

The C++ standard does not specify the error messages that compilers use to indicate particular errors. For this reason, we show in Fig. 15.7 the error messages produced by several compilers when a reference is not initialized.

### 15.8 Empty Parameter Lists

C++, like C, allows you to define functions with no parameters. In C++, an empty parameter list is specified by writing either `void` or nothing at all in parentheses. The prototypes

```cpp
void print();
void print( void );
```

each specify that function `print` does not take arguments and does not return a value. These prototypes are equivalent.

**Portability Tip 15.2**

The meaning of an empty function parameter list in C++ is dramatically different than in C. In C, it means all argument checking is disabled (i.e., the function call can pass any arguments it wants). In C++, it means that the function takes no arguments. Thus, C programs using this feature might cause compilation errors when compiled in C++.

### 15.9 Default Arguments

It’s not uncommon for a program to invoke a function repeatedly with the same argument value for a particular parameter. In such cases, the programmer can specify that such a parameter has a **default argument**, i.e., a default value to be passed to that parameter. When a program omits an argument for a parameter with a default argument in a function call, the compiler rewrites the function call and inserts the default value of that argument to be passed as an argument in the function call.

Default arguments must be the rightmost (trailing) arguments in a function’s parameter list. When calling a function with two or more default arguments, if an omitted argument is not the rightmost argument in the argument list, then all arguments to the right of that argument also must be omitted. Default arguments should be specified with the first occurrence of the function name—typically, in the function prototype. If the function prototype is omitted because the function definition also serves as the prototype, then the default arguments should be specified in the function header. Default values can be any expression, including constants, global variables or function calls. Default arguments also can be used with `inline` functions.

Figure 15.8 demonstrates using default arguments in calculating the volume of a box. The function prototype for `boxVolume` (line 7) specifies that all three parameters have been given default values of 1. We provided variable names in the function prototype for readability, but these are not required.
// Fig. 15.8: fig15_08.cpp
// Using default arguments.
#include <iostream>
using namespace std;

// function prototype that specifies default arguments
int boxVolume( int length = 1, int width = 1, int height = 1 );

int main()
{
    // no arguments--use default values for all dimensions
    cout << "The default box volume is: " << boxVolume();

    // specify length; default width and height
    cout << "\n\nThe volume of a box with length 10,\n" << "width 1 and height 1 is: " << boxVolume( 10 );

    // specify length and width; default height
    cout << "\n\nThe volume of a box with length 10,\n" << "width 5 and height 1 is: " << boxVolume( 10, 5 );

    // specify all arguments
    cout << "\n\nThe volume of a box with length 10,\n" << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
    << endl;
} // end main

// function boxVolume calculates the volume of a box
int boxVolume( int length, int width, int height )
{
    return length * width * height;
} // end function boxVolume

The default box volume is: 1
The volume of a box with length 10, width 1 and height 1 is: 10
The volume of a box with length 10, width 5 and height 1 is: 50
The volume of a box with length 10, width 5 and height 2 is: 100

Common Programming Error 15.6
It's a compilation error to specify default arguments in both a function's prototype and header.

The first call to boxVolume (line 12) specifies no arguments, thus using all three default values of 1. The second call (line 16) passes a length argument, thus using default values of 1 for the width and height arguments. The third call (line 20) passes arguments
for length and width, thus using a default value of 1 for the height argument. The last call (line 24) passes arguments for length, width and height, thus using no default values. Any arguments passed to the function explicitly are assigned to the function’s parameters from left to right. Therefore, when boxVolume receives one argument, the function assigns the value of that argument to its length parameter (i.e., the leftmost parameter in the parameter list). When boxVolume receives two arguments, the function assigns the values of those arguments to its length and width parameters in that order. Finally, when boxVolume receives all three arguments, the function assigns the values of those arguments to its length, width and height parameters, respectively.

Good Programming Practice 15.2
Using default arguments can simplify writing function calls. However, some programmers feel that explicitly specifying all arguments is clearer.

Software Engineering Observation 15.9
If the default values for a function change, all client code must be recompiled.

Common Programming Error 15.7
In a function definition, specifying and attempting to use a default argument that is not a rightmost (trailing) argument (while not simultaneously defaulting all the rightmost arguments) is a syntax error.

15.10 Unary Scope Resolution Operator

It’s possible to declare local and global variables of the same name. This causes the global variable to be “hidden” by the local variable in the local scope. C++ provides the unary scope resolution operator (::) to access a global variable when a local variable of the same name is in scope. The unary scope resolution operator cannot be used to access a local variable of the same name in an outer block. A global variable can be accessed directly without the unary scope resolution operator if the name of the global variable is not the same as that of a local variable in scope.

Figure 15.9 demonstrates the unary scope resolution operator with global and local variables of the same name (lines 6 and 10, respectively). To emphasize that the local and global versions of variable number are distinct, the program declares one variable of type int and the other double.

Good Programming Practice 15.2
Using default arguments can simplify writing function calls. However, some programmers feel that explicitly specifying all arguments is clearer.

Software Engineering Observation 15.9
If the default values for a function change, all client code must be recompiled.

Common Programming Error 15.7
In a function definition, specifying and attempting to use a default argument that is not a rightmost (trailing) argument (while not simultaneously defaulting all the rightmost arguments) is a syntax error.

### 15.10 Unary Scope Resolution Operator

It’s possible to declare local and global variables of the same name. This causes the global variable to be “hidden” by the local variable in the local scope. C++ provides the unary scope resolution operator (::) to access a global variable when a local variable of the same name is in scope. The unary scope resolution operator cannot be used to access a local variable of the same name in an outer block. A global variable can be accessed directly without the unary scope resolution operator if the name of the global variable is not the same as that of a local variable in scope.

Figure 15.9 demonstrates the unary scope resolution operator with global and local variables of the same name (lines 6 and 10, respectively). To emphasize that the local and global versions of variable number are distinct, the program declares one variable of type int and the other double.

```cpp
// Fig. 15.9: fig15_09.cpp
#include <iostream>
using namespace std;

int number = 7;  // global variable named number
double number = 10.5;  // local variable named number
```

---

Fig. 15.9 | Using the unary scope resolution operator. (Part 1 of 2.)
15.11 Function Overloading

C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least as far as the parameter types or the number of parameters or the order of the parameter types are concerned). This capability is called function overloading. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call.

1. The C++ standard requires float, double and long double overloaded versions of the math library functions discussed in Section 5.3.
Function overloading is commonly used to create several functions of the same name that perform similar tasks, but on data of different types. For example, many functions in the math library are overloaded for different numeric data types.

**Good Programming Practice 15.4**
Overloading functions that perform closely related tasks can make programs more readable and understandable.

**Overloaded square Functions**

Figure 15.10 uses overloaded square functions to calculate the square of an int (lines 7–11) and the square of a double (lines 14–18). Line 22 invokes the int version of function square by passing the literal value 7. C++ treats whole-number literal values as type int by default. Similarly, line 24 invokes the double version of function square by passing the literal value 7.5, which C++ treats as a double value by default. In each case the compiler chooses the proper function to call, based on the type of the argument. The outputs confirm that the proper function was called in each case.

```cpp
// Fig. 15.10: fig15_10.cpp
// Overloaded functions.
#include <iostream>
using namespace std;

// function square for int values
int square(int x)
{
    cout << "square of integer " << x << " is ";
    return x * x;
} // end function square with int argument

// function square for double values
double square(double y)
{
    cout << "square of double " << y << " is ";
    return y * y;
} // end function square with double argument

int main()
{
    cout << square(7); // calls int version
    cout << endl;
    cout << square(7.5); // calls double version
    cout << endl;
} // end main
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

*Fig. 15.10*  |  Overloaded square functions.

**How the Compiler Differentiates Overloaded Functions**
Overloaded functions are distinguished by their *signatures*—a combination of a function’s name and its parameter types (in order). The compiler encodes each function iden-
tifier with the number and types of its parameters (sometimes referred to as name mangling or name decoration) to enable type-safe linkage. This ensures that the proper overloaded function is called and that the argument types conform to the parameter types.

Figure 15.11 was compiled with GNU C++. Rather than showing the execution output of the program (as we normally would), we show the mangled function names produced in assembly language by GNU C++. Each mangled name (other than main) begins with two underscores (__) followed by the letter Z, a number and the function name. The number that follows Z specifies how many characters are in the function’s name. For example, function square has 6 characters in its name, so its mangled name is prefixed with __Z6. The function name is then followed by an encoding of its parameter list. In the parameter list for function nothing2 (line 25; see the fourth output line), c represents a char, i represents an int, Rf represents a float & (i.e., a reference to a float) and Rd represents a double & (i.e., a reference to a double). In the parameter list for function nothing1, i represents an int, f represents a float, c represents a char and Ri represents an int &. The two square functions are distinguished by their parameter lists; one specifies d for double and the other specifies i for int. The return types of the functions are not specified in the mangled names. Overloaded functions can have different return types, but if they do, they must also have different parameter lists. Again, you cannot have two functions with the same signature and different return types. Function-name mangling is compiler specific. Also, function main is not mangled, because it cannot be overloaded.

```cpp
1 // Fig. 15.11: fig15_11.cpp
2 // Name mangling.
3
4 // function square for int values
5 int square( int x )
6 {
7    return x * x;
8 } // end function square
9
10 // function square for double values
11 double square( double y )
12 {
13    return y * y;
14 } // end function square
15
16 // function that receives arguments of types
17 // int, float, char and int &
18 void nothing1( int a, float b, char c, int &d )
19 {
20    // empty function body
21 } // end function nothing1
22
23 // function that receives arguments of types
24 // char, int, float & and double &
25 int nothing2( char a, int b, float &c, double &d )
26 {
27    return 0;
28 } // end function nothing2
```

**Fig. 15.11** | Name mangling to enable type-safe linkage. (Part 1 of 2.)
The compiler uses only the parameter lists to distinguish between functions of the same name. Overloaded functions need not have the same number of parameters. Programmers should use caution when overloading functions with default parameters, because this may cause ambiguity.

**Overloaded Operators**

In Chapter 19, we discuss how to overload operators to define how they should operate on objects of user-defined data types. (In fact, we’ve been using overloaded operators, including the stream insertion operator `<<` and the stream extraction operator `>>`, each of which is overloaded to be able to display data of all the fundamental types. We say more about overloading `<<` and `>>` to be able to handle objects of user-defined types in Chapter 19.) Section 15.12 introduces function templates for automatically generating overloaded functions that perform identical tasks on data of different types.

### 15.12 Function Templates

Overloaded functions are used to perform similar operations that may involve different program logic on different data types. If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates. The programmer writes a single function template definition. Given the argument types provided in calls to this function, C++ automatically generates separate function template specializations to handle each type of call appropriately. Thus, defining a single function template essentially defines a whole family of overloaded functions.

```cpp
int main()
{
    return 0; // indicates successful termination
}
```

**Fig. 15.11 |** Name mangling to enable type-safe linkage. (Part 2 of 2.)

### Common Programming Error 15.9

Creating overloaded functions with identical parameter lists and different return types is a compilation error.

The compiler uses only the parameter lists to distinguish between functions of the same name. Overloaded functions need not have the same number of parameters. Programmers should use caution when overloading functions with default parameters, because this may cause ambiguity.

### Common Programming Error 15.10

A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error. For example, having in a program both a function that explicitly takes no arguments and a function of the same name that contains all default arguments results in a compilation error when an attempt is made to use that function name in a call passing no arguments. The compiler does not know which version of the function to choose.
Figure 15.12 contains the definition of a function template (lines 4–18) for a maximum function that determines the largest of three values. All function template definitions begin with the template keyword (line 4) followed by a template parameter list to the function template enclosed in angle brackets (< and >). Every parameter in the template parameter list (each is referred to as a formal type parameter) is preceded by keyword typename or keyword class (which are synonyms). The formal type parameters are placeholders for fundamental types or user-defined types. These placeholders are used to specify the types of the function’s parameters (line 5), to specify the function’s return type (line 5) and to declare variables within the body of the function definition (line 7). A function template is defined like any other function, but uses the formal type parameters as placeholders for actual data types.

The function template in Fig. 15.12 declares a single formal type parameter T (line 4) as a placeholder for the type of the data to be tested by function maximum. The name of a type parameter must be unique in the template parameter list for a particular template definition. When the compiler detects a maximum invocation in the program source code, the type of the data passed to maximum is substituted for T throughout the template definition, and C++ creates a complete source-code function for determining the maximum of three values of the specified data type. Then the newly created function is compiled. Thus, templates are a means of code generation.

```cpp
// Fig. 15.12: maximum.h
// Definition of function template maximum.

template < class T > // or template< typename T >
T maximum( T value1, T value2, T value3 )
{
    T maximumValue = value1; // assume value1 is maximum
    if ( value2 > maximumValue )
        maximumValue = value2;
    if ( value3 > maximumValue )
        maximumValue = value3;
    return maximumValue;
} // end function template maximum
```

**Fig. 15.12** | Function template maximum header file.

**Common Programming Error 15.11**

*Not placing keyword class or keyword typename before every formal type parameter of a function template (e.g., writing < class S, T > instead of < class S, class T >) is a syntax error.*

Figure 15.13 uses the maximum function template (lines 18, 28 and 38) to determine the largest of three int values, three double values and three char values.
In Fig. 15.13, three functions are created as a result of the calls in lines 18, 28 and 38—expecting three int values, three double values and three char values, respectively. For example, the function template specialization created for type int replaces each occurrence of T with int as follows:
15.13 Introduction to Object Technology and the UML

Now we introduce object orientation, a natural way of thinking about the world and writing computer programs. Our goal here is to help you develop an object-oriented way of thinking and to introduce you to the Unified Modeling Language™ (UML™)—a graphical language that allows people who design object-oriented software systems to use an industry-standard notation to represent them. In this section, we introduce basic object-oriented concepts and terminology.

Basic Object Technology Concepts

We begin our introduction to object orientation with some key terminology. Everywhere you look in the real world you see objects—people, animals, plants, cars, planes, buildings, computers and so on. Humans think in terms of objects. Telephones, houses, traffic lights, microwave ovens and water coolers are just a few more objects we see around us every day.

Objects have some things in common. They all have attributes (e.g., size, shape, color and weight), and they all exhibit behaviors (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleeps, crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water). We’ll study the kinds of attributes and behaviors that software objects have.

Humans learn about existing objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults and between humans and chimpanzees.

Object-oriented design (OOD) models software in terms similar to those that people use to describe real-world objects. It takes advantage of class relationships, where objects of a certain class, such as a class of vehicles, have the same characteristics—cars, trucks, little red wagons and roller skates have much in common. OOD takes advantage of inheritance relationships, where new classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. An object of class “convertible” certainly has the characteristics of the more general class “automobile,” but more specifically, the roof goes up and down.

Object-oriented design provides a natural and intuitive way to view the software design process—namely, modeling objects by their attributes, behaviors and interrelationships just as we describe real-world objects. OOD also models communication between objects. Just as people send messages to one another (e.g., a sergeant commands a soldier to stand at attention), objects also communicate via messages. A bank account object may...
receive a message to decrease its balance by a certain amount because the customer has
withdrawn that amount of money.

OOD encapsulates (i.e., wraps) attributes and operations (behaviors) into objects—an object’s attributes and operations are intimately tied together. Objects have the property of information hiding. This means that objects may know how to communicate with one another across well-defined interfaces, but normally they’re not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves. We can drive a car effectively, for instance, without knowing the details of how engines, transmissions, brakes and exhaust systems work internally—as long as we know how to use the accelerator pedal, the brake pedal, the steering wheel and so on. Information hiding, as we’ll see, is crucial to good software engineering.

Languages like C++ are object oriented. Programming in such a language is called object-oriented programming (OOP), and it allows you to implement an object-oriented design as a working software system. Languages like C, on the other hand, are procedural, so programming tends to be action oriented. In C, the unit of programming is the function. In C++, the unit of programming is the “class” from which objects are eventually instantiated (an OOP term for “created”). C++ classes contain functions that implement operations and data that implements attributes.

C programmers concentrate on writing functions. Programmers group actions that perform some common task into functions, and group functions to form programs. Data is certainly important in C, but the view is that data exists primarily in support of the actions that functions perform. The verbs in a system specification help the C programmer determine the set of functions that will work together to implement the system.

**Classes, Data Members and Member Functions**

C++ programmers concentrate on creating their own user-defined types called classes. Each class contains data as well as the set of functions that manipulate that data and provide services to clients (i.e., other classes or functions that use the class). The data components of a class are called data members. For example, a bank account class might include an account number and a balance. The function components of a class are called member functions (typically called methods in other object-oriented programming languages such as Java). For example, a bank account class might include member functions to make a deposit (increasing the balance), make a withdrawal (decreasing the balance) and inquire what the current balance is. The programmer uses built-in types (and other user-defined types) as the “building blocks” for constructing new user-defined types (classes). The nouns in a system specification help the C++ programmer determine the set of classes from which objects are created that work together to implement the system.

Classes are to objects as blueprints are to houses—a class is a “plan” for building an object of the class. Just as we can build many houses from one blueprint, we can instantiate (create) many objects from one class. You cannot cook meals in the kitchen of a blueprint; you can cook meals in the kitchen of a house. You cannot sleep in the bedroom of a blueprint; you can sleep in the bedroom of a house.

Classes can have relationships with other classes. In an object-oriented design of a bank, the “bank teller” class relates to other classes, such as the “customer” class, the “cash drawer” class, the “safe” class, and so on. These relationships are called associations. Packaging software as classes makes it possible for future software systems to reuse the classes.
15.13 Introduction to Object Technology and the UML

Indeed, with object technology, you can build much of the new software you’ll need by combining existing classes, just as automobile manufacturers combine interchangeable parts. Each new class you create can become a valuable software asset that you and others can reuse to speed and enhance the quality of future software development efforts.

Introduction to Object-Oriented Analysis and Design (OOAD)

Soon you’ll be writing programs in C++. How will you create the code for your programs? Perhaps, like many beginning programmers, you’ll simply turn on your computer and start typing. This approach may work for small programs, but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or what if you were asked to work on a team of 1000 software developers building the next generation of the U.S. air traffic control system? For projects so large and complex, you could not simply sit down and start writing programs.

To create the best solutions, you should follow a detailed process for analyzing your project’s requirements (i.e., determining what the system is supposed to do) and developing a design that satisfies them (i.e., deciding how the system should do it). Ideally, you would go through this process and carefully review the design (or have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it’s called object-oriented analysis and design (OOAD). Experienced programmers know that analysis and design can save many hours by helping avoid an ill-planned system development approach that has to be abandoned partway through its implementation, possibly wasting considerable time, money and effort.

OOAD is the generic term for the process of analyzing a problem and developing an approach for solving it. Small problems like the ones discussed in these first few chapters do not require an exhaustive OOAD process.

As problems and the groups of people solving them increase in size, the methods of OOAD quickly become more appropriate than pseudocode. Ideally, a group should agree on a strictly defined process for solving its problem and a uniform way of communicating the results of that process to one another. Although many different OOAD processes exist, a single graphical language for communicating the results of any OOAD process has come into wide use. This language, known as the Unified Modeling Language (UML), was developed in the mid-1990s under the initial direction of three software methodologists: Grady Booch, James Rumbaugh and Ivar Jacobson.

History of the UML

In the 1980s, increasing numbers of organizations began using OOP to build their applications, and a need developed for a standard OOAD process. Many methodologists—including Booch, Rumbaugh and Jacobson—individually produced and promoted separate processes to satisfy this need. Each process had its own notation, or “language” (in the form of graphical diagrams), to convey the results of analysis and design.
In 1994, James Rumbaugh joined Grady Booch at Rational Software Corporation (now a division of IBM), and the two began working to unify their popular processes. They soon were joined by Ivar Jacobson. In 1996, the group released early versions of the UML to the software engineering community and requested feedback. Around the same time, an organization known as the Object Management Group™ (OMG™) invited submissions for a common modeling language. The OMG (www.omg.org) is a nonprofit organization that promotes the standardization of object-oriented technologies by issuing guidelines and specifications, such as the UML. Several corporations—among them HP, IBM, Microsoft, Oracle and Rational Software—had already recognized the need for a common modeling language. In response to the OMG’s request for proposals, these companies formed UML Partners—the consortium that developed the UML version 1.1 and submitted it to the OMG. The OMG accepted the proposal and, in 1997, assumed responsibility for the continuing maintenance and revision of the UML. We present the terminology and notation of the current version of the UML—UML version 2—throughout the C++ section of this book.

What Is the UML?
The Unified Modeling Language is now the most widely used graphical representation scheme for modeling object-oriented systems. Those who design systems use the language (in the form of diagrams) to model their systems, as we do throughout the C++ section of this book. An attractive feature of the UML is its flexibility. The UML is extensible (i.e., capable of being enhanced with new features) and is independent of any particular OOAD process. UML modelers are free to use various processes in designing systems, but all developers can now express their designs with one standard set of graphical notations. For more information, visit our UML Resource Center at www.deitel.com/UML/.

15.14 Wrap-Up

In this chapter, you learned several of C++’s enhancements to C. We presented basic C++-style input and output with cin and cout and overviewed the C++ Standard Library header files. We discussed inline functions for improving performance by eliminating the overhead of function calls. You learned how to use pass-by-reference with C++’s reference parameters, which enable you to create aliases for existing variables. You learned that multiple functions can be overloaded by providing functions with the same name and different signatures; such functions can be used to perform the same or similar tasks, using different types or different numbers of parameters. We then demonstrated a simpler way of overloading functions using function templates, where a function is defined once but can be used for several different types. You learned the basic terminology of object technology and were introduced to the UML—the most widely used graphical representation scheme for modeling OO systems. In Chapter 16, you’ll learn how to implement your own classes and use objects of those classes in applications.

Summary

Section 15.2 C++

- C++ improves on many of C’s features and provides object-oriented-programming (OOP) capabilities that increase software productivity, quality and reusability.
• C++ was developed by Bjarne Stroustrup at Bell Labs and was originally called “C with classes.”

Section 15.3 A Simple Program: Adding Two Integers

• C++ filenames can have one of several extensions, such as .cpp, .cxx or .C (uppercase).

• C++ allows you to begin a comment with // and use the remainder of the line as comment text. C++ programmers may also use C-style comments.

• The input/output stream header file <iostream> must be included for any program that outputs data to the screen or inputs data from the keyboard using C++-style stream input/output.

• As in C, every C++ program begins execution with function main. Keyword int to the left of main indicates that main “returns” an integer value.

• In C, you need not specify a return type for functions. However, C++ requires you to specify the return type, possibly void, for all functions; otherwise, a syntax error occurs.

• Declarations can be placed almost anywhere in a C++ program, but they must appear before their corresponding variables are used in the program.

• The standard output stream object (std::cout) and the stream insertion operator (<<) are used to display text on the screen.

• The standard input stream object (std::cin) and the stream extraction operator (>>) are used to obtain values from the keyboard.

• The stream manipulator std::endl outputs a newline, then “flushes the output buffer.”

• The notation std::cout specifies that we’re using a name, in this case cout, that belongs to “namespace” std.

• Using multiple stream insertion operators (<<) in a single statement is referred to as concatenating, chaining or cascading stream insertion operations.

Section 15.4 C++ Standard Library

• C++ programs consist of pieces called classes and functions. You can program each piece you may need to form a C++ program. However, most C++ programmers take advantage of the rich collections of existing classes and functions in the C++ Standard Library.

Section 15.5 Header Files

• The C++ Standard Library is divided into many portions, each with its own header file. The header files contain the function prototypes for the related functions that form each portion of the library. The header files also contain definitions of various class types and functions, as well as constants needed by those functions.

• Header file names ending in .h are “old-style” header files that have been superceded by the C++ Standard Library header files.

Section 15.6 Inline Functions

• C++ provides inline functions to help reduce function call overhead—especially for small functions. Placing the qualifier inline before a function’s return type in the function definition “advises” the compiler to generate a copy of the function’s code in place to avoid a function call.

Section 15.7 References and Reference Parameters

• Two ways to pass arguments to functions in many programming languages are pass-by-value and pass-by-reference.

• When an argument is passed by value, a copy of its value is made and passed (on the function call stack) to the called function. Changes to the copy do not affect the original in the caller.

• With pass-by-reference, the caller gives the called function the ability to access the caller’s data directly and to modify it if the called function chooses to do so.
• A reference parameter is an alias for its corresponding argument in a function call.
• To indicate that a function parameter is passed by reference, simply follow the parameter’s type in the function prototype by an ampersand (&); use the same notation when listing the parameter’s type in the function header.
• Once a reference is declared as an alias for another variable, all operations supposedly performed on the alias (i.e., the reference) are actually performed on the original variable. The alias is simply another name for the original variable.

Section 15.8 Empty Parameter Lists
• In C++, an empty parameter list is specified by writing either void or nothing in parentheses.

Section 15.9 Default Arguments
• It’s not uncommon for a program to invoke a function repeatedly with the same argument value for a particular parameter. In such cases, the programmer can specify that such a parameter has a default argument, i.e., a default value to be passed to that parameter.
• When a program omits an argument for a parameter with a default argument, the compiler inserts the default value of that argument to be passed as an argument in the function call.
• Default arguments must be the rightmost (trailing) arguments in a function’s parameter list.
• Default arguments should be specified with the first occurrence of the function name—typically, in the function prototype.

Section 15.10 Unary Scope Resolution Operator
• C++ provides the unary scope resolution operator (::) to access a global variable when a local variable of the same name is in scope.

Section 15.11 Function Overloading
• C++ enables several functions of the same name to be defined, as long as they have different sets of parameters (by number, type and/or order). This capability is called function overloading.
• When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call.
• Overloaded functions are distinguished by their signatures.
• The compiler encodes each function identifier with the number and types of its parameters to enable type-safe linkage. Type-safe linkage ensures that the proper overloaded function is called and that the types of the arguments conform to the types of the parameters.

Section 15.12 Function Templates
• Overloaded functions are used to perform similar operations that may involve different program logic on data of different types. If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently using function templates.
• The programmer writes a single function template definition. Given the argument types provided in calls to this function, C++ automatically generates separate function template specializations to handle each type of call appropriately. Thus, defining a single function template essentially defines a family of overloaded functions.
• All function template definitions begin with the template keyword followed by a template parameter list to the function template enclosed in angle brackets (< and >).
• The formal type parameters are placeholders for fundamental types or user-defined types. These placeholders are used to specify the types of the function’s parameters, to specify the function’s return type and to declare variables within the body of the function definition.
Section 15.13 Introduction to Object Technology and the UML

- The Unified Modeling Language (UML) is a graphical language that allows people who build systems to represent their object-oriented designs in a common notation.

- Object-oriented design (OOD) models software components in terms of real-world objects. It takes advantage of class relationships, where objects of a certain class have the same characteristics. It also takes advantage of inheritance relationships, where newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. OOD encapsulates data (attributes) and functions (behavior) into objects—the data and functions of an object are intimately tied together.

- Objects have the property of information hiding—objects normally are not allowed to know how other objects are implemented.

- Object-oriented programming (OOP) allows programmers to implement object-oriented designs as working systems.

- C++ programmers create their own user-defined types called classes. Each class contains data (known as data members) and the set of functions (known as member functions) that manipulate that data and provide services to clients.

- Classes can have relationships with other classes. These relationships are called associations.

- Packaging software as classes makes it possible for future software systems to reuse the classes. Groups of related classes are often packaged as reusable components.

- An instance of a class is called an object.

- With object technology, programmers can build much of the software they will need by combining standardized, interchangeable parts called classes.

- The process of analyzing and designing a system from an object-oriented point of view is called object-oriented analysis and design (OOAD).

**Terminology**

::, unary scope resolution operator 544
action oriented 552
analyze a requirements document 553
association (in the UML) 552
attribute of an object 551
behavior of an object 551
bool keyword 536
C++ Standard Library 532
cascading stream insertion operations 532
chaining stream insertion operations 532
class 532
client of a class 552
concatenating stream insertion operations 532
dangling reference 541
data member of a class 552
default argument 542
design a system 553
encapsulation 552
end1 stream manipulator 531
extensible language 554
false keyword 536
formal type parameter 549
function overloading 545
function template 548
function template specialization 548
generic programming 529
information hiding 552
inheritance 551
inline function 535
inline keyword 535
input/output stream header (<iostream>) 530
mangled function name 547
member function 552
method of a class 552
name decoration 547
name mangling 547
object 551
Object Management Group (OMG) 554
object-oriented analysis and design (OOAD) 553
object-oriented design (OOD) 551
object-oriented language 552
object-oriented programming (OOP) 529
operation of a class 552
operator overloading 532
procedural programming language 552
reference parameter 537
requirement of a project 553
reuse classes 552
signature of a function 546
standard input stream object (cin) 531
standard output stream object (cout) 531
stream extraction operator (>>) 531
stream insertion operator (<<) 531
stream manipulator 531
template parameter list 549
true keyword 536
type-safe linkage 538
Unified Modeling Language (UML) 551

Self-Review Exercises
15.1 Answer each of the following:
   a) In C++, it’s possible to have various functions with the same name that operate on different types or numbers of arguments. This is called function ________.
   b) The ________ enables access to a global variable with the same name as a variable in the current scope.
   c) A function ________ enables a single function to be defined to perform the same task on data of many different types.
   d) ________ is the most widely used graphical representation scheme for OO modeling.
   e) ________ models software components in terms of real-world objects.
   f) C++ programmers create their own user-defined types called ________.

15.2 Why would a function prototype contain a parameter type declaration such as double &?
15.3 (True/False) All arguments to function calls in C++ are passed by value.
15.4 Write a complete program that prompts the user for the radius of a sphere, and calculates and prints the volume of that sphere. Use an inline function sphereVolume that returns the result of the following expression: \((4.0 / 3.0) \times 3.14159 \times \text{pow}(\text{radius}, 3)\).

Answers to Self-Review Exercises
15.1 a) overloading. b) unary scope resolution operator (::). c) template. d) The UML. e) Object-oriented design (OOD). f) classes.
15.2 This creates a reference parameter of type “reference to double” that enables the function to modify the original variable in the calling function.
15.4 See the following program:

```cpp
// Exercise 15.4 Solution: Ex15_04.cpp
// Inline function that calculates the volume of a sphere.
#include <iostream>
#include <cmath>
const double PI = 3.14159; // define global constant PI

// calculates volume of a sphere
inline double sphereVolume( const double radius )
{
    return 4.0 / 3.0 * PI * pow( radius, 3 );
} // end inline function sphereVolume

int main() {
```
Exercises

15.5 Write a C++ program that prompts the user for the radius of a circle, then calls inline function circleArea to calculate the area of that circle.

15.6 Write a complete C++ program with the two alternate functions specified below, each of which simply triples the variable count defined in main. Then compare and contrast the two approaches. These two functions are

   a) function tripleByValue that passes a copy of count by value, triples the copy and returns the new value and
   a) function tripleByReference that passes count by reference via a reference parameter and triples the original value of count through its alias (i.e., the reference parameter).

15.7 What is the purpose of the unary scope resolution operator?

15.8 Write a program that uses a function template called min to determine the smaller of two arguments. Test the program using integer, character and floating-point number arguments.

15.9 Write a program that uses a function template called max to determine the larger of two arguments. Test the program using integer, character and floating-point number arguments.

15.10 Determine whether the following program segments contain errors. For each error, explain how it can be corrected. [Note: For a particular program segment, it’s possible that no errors are present in the segment.]

   a) template < class A >
      int sum( int num1, int num2, int num3 )
      {
         return num1 + num2 + num3;
      }
   b) void printResults( int x, int y )
      {
         cout << "The sum is " << x + y << '\n';
         return x + y;
      }
   c) template < A >
      A product( A num1, A num2, A num3 )
      {
         return num1 * num2 * num3;
      }
   d) double cube( int );
      int cube( int );
Objectives

In this chapter you’ll learn:

- How to define a class and use it to create an object.
- How to define member functions in a class to implement the class’s behaviors.
- How to declare data members in a class to implement the class’s attributes.
- How to call a member function of an object to perform a task.
- The differences between data members of a class and local variables of a function.
- How to use a constructor to initialize an object’s data when the object is created.
- How to engineer a class to separate its interface from its implementation and encourage reuse.
16.1 Introduction

In this chapter, you'll begin writing programs that employ the basic concepts of object-oriented programming that we introduced in Section 15.13. Typically, the programs you develop in C++ will consist of function main and one or more classes, each containing data members and member functions. If you become part of a development team in industry, you might work on software systems that contain hundreds, or even thousands, of classes. In this chapter, we develop a simple, well-engineered framework for organizing object-oriented programs in C++.

First, we motivate the notion of classes with a real-world example. Then we present a carefully paced sequence of seven complete working programs to demonstrate creating and using your own classes.

16.2 Classes, Objects, Member Functions and Data Members

Let’s begin with a simple analogy to help you reinforce your understanding from Section 15.13 of classes and their contents. Suppose you want to drive a car and make it go faster by pressing down on its accelerator pedal. What must happen before you can do this? Well, before you can drive a car, someone has to design it and build it. A car typically begins as engineering drawings, similar to the blueprints used to design a house. These drawings include the design for an accelerator pedal that the driver will use to make the car go faster. In a sense, the pedal “hides” the complex mechanisms that actually make the car go faster, just as the brake pedal “hides” the mechanisms that slow the car, the steering wheel “hides” the mechanisms that turn the car and so on. This enables people with little or no knowledge of how cars are engineered to drive a car easily, simply by using the accelerator pedal, the brake pedal, the steering wheel, the transmission shifting mechanism and other such simple and user-friendly “interfaces” to the car’s complex internal mechanisms.

Unfortunately, you cannot drive the engineering drawings of a car—before you can drive a car, it must be built from the engineering drawings that describe it. A completed car will have an actual accelerator pedal to make the car go faster. But even that’s not enough—the car will not accelerate on its own, so the driver must press the accelerator pedal to tell the car to go faster.
Now let’s use our car example to introduce the key object-oriented programming concepts of this section. Performing a task in a program requires a function (such as `main`). The function describes the mechanisms that actually perform its tasks. The function hides from its user the complex tasks that it performs, just as the accelerator pedal of a car hides from the driver the complex mechanisms of making the car go faster. In C++, we begin by creating a program unit called a class to house a function, just as a car’s engineering drawings house the design of an accelerator pedal. Recall from Section 15.13 that a function belonging to a class is called a member function. In a class, you provide one or more member functions that are designed to perform the class’s tasks. For example, a class that represents a bank account might contain one member function to deposit money into the account, another to withdraw money from the account and a third to inquire what the current account balance is.

Just as you cannot drive an engineering drawing of a car, you cannot “drive” a class. Just as someone has to build a car from its engineering drawings before you can actually drive the car, you must create an object of a class before you can get a program to perform the tasks the class describes. That is one reason C++ is known as an object-oriented programming language. Note also that just as many cars can be built from the same engineering drawing, many objects can be built from the same class.

When you drive a car, pressing its gas pedal sends a message to the car to perform a task—that is, make the car go faster. Similarly, you send messages to an object—each message is known as a member-function call and tells a member function of the object to perform its task. This is often called requesting a service from an object.

Thus far, we’ve used the car analogy to introduce classes, objects and member functions. In addition to the capabilities a car provides, it also has many attributes, such as its color, the number of doors, the amount of gas in its tank, its current speed and its total miles driven (i.e., its odometer reading). Like the car’s capabilities, these attributes are represented as part of a car’s design in its engineering diagrams. As you drive a car, these attributes are always associated with the car. Every car maintains its own attributes. For example, each car knows how much gas is in its own gas tank, but not how much is in the tanks of other cars. Similarly, an object has attributes that are carried with the object as it’s used in a program. These attributes are specified as part of the object’s class. For example, a bank account object has a balance attribute that represents the amount of money in the account. Each bank account object knows the balance in the account it represents, but not the balances of the other accounts in the bank. Attributes are specified by the class’s data members.

The remainder of this chapter presents seven simple examples that demonstrate the concepts we introduced in the context of the car analogy.

### 16.3 Defining a Class with a Member Function

We begin with an example (Fig. 16.1) that consists of class `GradeBook` (lines 8–16), which will represent a grade book that an instructor can use to maintain student test scores, and a `main` function (lines 19–23) that creates a `GradeBook` object. Function `main` uses this object and its member function to display a message on the screen welcoming the instructor to the grade-book program.

First we describe how to define a class and a member function, then how an object is created and how to call an object’s member function. The first few examples contain in
16.3 Defining a Class with a Member Function

Before function main (lines 19–23) can create a GradeBook object, we must tell the compiler what member functions and data members belong to the class—known as **defining a class**. The GradeBook class definition (lines 8–16) begins with keyword `class` and contains a member function called `displayMessage` (lines 12–15) that displays a message on the screen (line 14). Recall that a class is like a blueprint—so we need to make an object of class GradeBook (line 21) and call its `displayMessage` member function (line 22) to get line 14 to execute and display the welcome message. We'll soon explain lines 21–22 in detail.

The class definition begins in line 8 with the keyword `class` followed by the class name GradeBook. By convention, the name of a user-defined class begins with a capital letter, and for readability, each subsequent word in the class name begins with a capital letter. This capitalization style is often referred to as **camel case**, because the pattern of uppercase and lowercase letters resembles the silhouette of a camel.

Every class’s body is enclosed in a pair of left and right braces ({ and }), as in lines 9 and 16. The class definition terminates with a semicolon (line 16).
Recall that the function `main` is always called automatically when you execute a program. Most functions do not get called automatically. As you’ll soon see, you must call member function `displayMessage` explicitly to tell it to perform its task.

Line 10 contains the `access-specifier label public:`. The keyword `public` is an access specifier. Lines 12–15 define member function `displayMessage`. This member function appears after access specifier `public:` to indicate that the function is “available to the public”—that is, it can be called by other functions in the program (such as `main`), and by member functions of other classes (if there are any). Access specifiers are always followed by a colon (:``). For the remainder of the text, when we refer to the access specifier `public`, we’ll omit the colon as we did in this sentence. Section 16.5 introduces a second access specifier, `private`. Later in the book we’ll study the access specifier `protected`.

Each function in a program performs a task and may return a value when it completes its task—for example, a function might perform a calculation, then return the result of that calculation. When you define a function, you must specify a return type to indicate the type of the value returned by the function when it completes its task. In line 12, keyword `void` to the left of the function name `displayMessage` is the function’s return type. Return type `void` indicates that `displayMessage` will not return (i.e., give back) any data to its calling function (in this example, `main`, as we’ll see in a moment) when it completes its task. In Fig. 16.5, you’ll see an example of a function that returns a value.

The name of the member function, `displayMessage`, follows the return type. By convention, function names begin with a lowercase first letter and all subsequent words in the name begin with a capital letter. The parentheses after the member function name indicate that this is a function. An empty set of parentheses, as shown in line 12, indicates that this member function does not require additional data to perform its task. You’ll see an example of a member function that does require additional data in Section 16.4. Line 12 is commonly referred to as the function header. Every function’s body is delimited by left and right braces (`{` and `}`), as in lines 13 and 15.

The body of a function contains statements that perform the function’s task. In this case, member function `displayMessage` contains one statement (line 14) that displays the message "Welcome to the Grade Book!". After this statement executes, the function has completed its task.

**Common Programming Error 16.2**

*Returning a value from a function whose return type has been declared `void` is a compilation error.*

**Common Programming Error 16.3**

*Defining a function inside another function (i.e., “nesting” functions) is a syntax error.*

**Testing Class GradeBook**

Next, we’d like to use class `GradeBook` in a program. As you know, function `main` (lines 19–23) begins the execution of every program.

In this program, we’d like to call class `GradeBook`’s `displayMessage` member function to display the welcome message. Typically, you cannot call a member function of a class until you create an object of that class. (As you’ll learn in Section 18.6, static member
16.3 Defining a Class with a Member Function

functions are an exception.) Line 21 creates an object of class GradeBook called myGradeBook. The variable’s type is GradeBook—the class we defined in lines 8–16. When we declare variables of type int, the compiler knows what int is—it’s a fundamental type. In line 21, however, the compiler does not automatically know what type GradeBook is—it’s a **user-defined type**. We tell the compiler what GradeBook is by including the class definition (lines 8–16). If we omitted these lines, the compiler would issue an error message (such as “‘GradeBook’: undeclared identifier” in Microsoft Visual C++ or “‘GradeBook’: undeclared” in GNU C++). Each class you create becomes a new type that can be used to create objects. You can define new class types as needed; this is one reason why C++ is known as an **extensible language**.

Line 22 calls the member function `displayMessage` (defined in lines 12–15) using variable myGradeBook followed by the **dot operator** (.), the function name `displayMessage` and an empty set of parentheses. This call causes the `displayMessage` function to perform its task. At the beginning of line 22, “myGradeBook.” indicates that main should use the GradeBook object that was created in line 21. The empty parentheses in line 12 indicate that member function `displayMessage` does not require additional data to perform its task, which is why we called this function with empty parentheses in line 22. (In Section 16.4, you’ll see how to pass data to a function.) When `displayMessage` completes its task, the program reaches the end of main and terminates.

**UML Class Diagram for Class GradeBook**

Recall from Section 15.13 that the UML is a standardized graphical language used by software developers to represent their object-oriented systems. In the UML, each class is modeled in a **UML class diagram** as a rectangle with three compartments. Figure 16.2 presents a class diagram for class GradeBook (Fig. 16.1). The top compartment contains the class’s name centered horizontally and in boldface type. The middle compartment contains the class’s attributes, which correspond to data members in C++. This compartment is currently empty, because class GradeBook does not have any attributes. (Section 16.5 presents a version of class GradeBook with an attribute.) The bottom compartment contains the class’s operations, which correspond to member functions in C++. The UML models operations by listing the operation name followed by a set of parentheses. Class GradeBook has only one member function, `displayMessage`, so the bottom compartment of Fig. 16.2 lists one operation with this name. Member function `displayMessage` does not require additional information to perform its tasks, so the parentheses following `displayMessage` in the class diagram are empty, just as they are in the member function’s header in line 12 of Fig. 16.1. The plus sign (+) in front of the operation name indicates that `displayMessage` is a public operation in the UML (i.e., a public member function in C++).
Chapter 16  Introduction to Classes and Objects

16.4 Defining a Member Function with a Parameter

In our car analogy from Section 16.2, we mentioned that pressing a car’s gas pedal sends a message to the car to perform a task—make the car go faster. But how fast should the car accelerate? As you know, the farther down you press the pedal, the faster the car accelerates. So the message to the car includes both the task to perform and additional information that helps the car perform the task. This additional information is known as a parameter—the value of the parameter helps the car determine how fast to accelerate. Similarly, a member function can require one or more parameters that represent additional data it needs to perform its task. A function call supplies values—called arguments—for each of the function’s parameters. For example, to make a deposit into a bank account, suppose a deposit member function of an Account class specifies a parameter that represents the deposit amount. When the deposit member function is called, an argument value representing the deposit amount is copied to the member function’s parameter. The member function then adds that amount to the account balance.

Defining and Testing Class GradeBook

Our next example (Fig. 16.3) redefines class GradeBook (lines 9–18) with a displayMessage member function (lines 13–17) that displays the course name as part of the welcome message. The new version of displayMessage requires a parameter (courseName in line 13) that represents the course name to output.

```cpp
// Fig. 16.3: fig16_03.cpp
// Define class GradeBook with a member function that takes a parameter;
// create a GradeBook object and call its displayMessage function.
#include <iostream>
#include <string> // program uses C++ standard string class
using namespace std;

// GradeBook class definition
class GradeBook
{
 public:
   // function that displays a welcome message to the GradeBook user
   void displayMessage( string courseName )
   {
      cout << "Welcome to the grade book for\n" << courseName << "!"
   }

}; // end class GradeBook

// function main begins program execution
int main()
{
   string nameOfCourse; // string of characters to store the course name
   GradeBook myGradeBook; // create a GradeBook object named myGradeBook

   Fig. 16.3  |  Define class GradeBook with a member function that takes a parameter, create a GradeBook object and call its displayMessage function. (Part 1 of 2.)
```
Before discussing the new features of class `GradeBook`, let’s see how the new class is used in `main` (lines 21–34). Line 23 creates a variable of type `string` called `nameOfCourse` that will be used to store the course name entered by the user. A variable of type string represents a string of characters such as “CS101 Introduction to C++ Programming”. A string is actually an object of the C++ Standard Library class `string`. This class is defined in header file `<string>`, and the name string, like `cout`, belongs to namespace `std`. To enable line 23 to compile, line 5 includes the `<string>` header file. The using declaration in line 6 allows us to simply write `string` in line 23 rather than `std::string`. For now, you can think of string variables like variables of other types such as `int`. You’ll learn additional string capabilities in Section 16.9.

Line 24 creates an object of class `GradeBook` named `myGradeBook`. Line 27 prompts the user to enter a course name. Line 28 reads the name from the user and assigns it to the `nameOfCourse` variable, using the library function `getline` to perform the input. Before we explain this line of code, let’s explain why we cannot simply write

```cpp
  cin >> nameOfCourse;
```

to obtain the course name. In our sample program execution, we use the course name “CS101 Introduction to C++ Programming,” which contains multiple words. (Recall that we highlight user-supplied input in bold.) When `cin` is used with the stream extraction operator, it reads characters until the first white-space character is reached. Thus, only “CS101” would be read by the preceding statement. The rest of the course name would have to be read by subsequent input operations.

In this example, we’d like the user to type the complete course name and press `Enter` to submit it to the program, and we’d like to store the entire course name in the string variable `nameOfCourse`. The function call `getline(cin, nameOfCourse)` in line 28 reads characters (including the space characters that separate the words in the input) from the standard input stream object `cin` (i.e., the keyboard) until the newline character is encountered, places the characters in the string variable `nameOfCourse` and discards the newline character. When you press `Enter` while typing program input, a newline is inserted in the
input stream. Also, the <string> header file must be included in the program to use function getline and that the name getline belongs to namespace std.

Line 33 calls myGradeBook’s displayMessage member function. The nameOfCourse variable in parentheses is the argument that is passed to member function displayMessage so that it can perform its task. The value of variable nameOfCourse in main becomes the value of member function displayMessage’s parameter courseName in line 13. When you execute this program, member function displayMessage outputs as part of the welcome message the course name you type (in our sample execution, CS101 Introduction to C++ Programming).

**More on Arguments and Parameters**

To specify that a function requires data to perform its task, you place additional information in the function’s **parameter list**, which is located in the parentheses following the function name. The parameter list may contain any number of parameters, including none at all (represented by empty parentheses as in Fig. 16.1, line 12) to indicate that a function does not require any parameters. Member function displayMessage’s parameter list (Fig. 16.3, line 13) declares that the function requires one parameter. Each parameter must specify a type and an identifier. In this case, the type string and the identifier courseName indicate that member function displayMessage requires a string to perform its task. The member function body uses the parameter courseName to access the value that is passed to the function in the function call (line 33 in main). Lines 15–16 display parameter courseName’s value as part of the welcome message. The parameter variable’s name (line 13) can be the same as or different from the argument variable’s name (line 33).

A function can specify multiple parameters by separating each parameter from the next with a comma. The number and order of arguments in a function call must match the number and order of parameters in the parameter list of the called member function’s header. Also, the argument types in the function call must be consistent with the types of the corresponding parameters in the function header. (As you’ll learn in subsequent chapters, an argument’s type and its corresponding parameter’s type need not always be identical, but they must be “consistent.”) In our example, the one string argument in the function call (i.e., nameOfCourse) exactly matches the one string parameter in the member-function definition (i.e., courseName).

---

**Common Programming Error 16.4**

Placing a semicolon after the right parenthesis enclosing the parameter list of a function definition is a syntax error.

**Common Programming Error 16.5**

Defining a function parameter again as a variable in the function’s body is a compilation error.

**Good Programming Practice 16.1**

To avoid ambiguity, do not use the same names for the arguments passed to a function and the corresponding parameters in the function definition.
Updated UML Class Diagram for Class GradeBook

The UML class diagram of Fig. 16.4 models class GradeBook of Fig. 16.3. Like the class GradeBook defined in Fig. 16.1, this GradeBook class contains public member function displayMessage. However, this version of displayMessage has a parameter. The UML models a parameter by listing the parameter name, followed by a colon and the parameter type in the parentheses following the operation name. The UML has its own data types similar to those of C++. The UML is language independent—it’s used with many different programming languages—so its terminology does not exactly match that of C++. For example, the UML type String corresponds to the C++ type string. Member function displayMessage of class GradeBook (Fig. 16.3, lines 13–17) has a string parameter named courseName, so Fig. 16.4 lists courseName : String between the parentheses following the operation name displayMessage. This version of the GradeBook class still does not have any data members.

Fig. 16.4 | UML class diagram indicating that class GradeBook has a public displayMessage operation with a courseName parameter of UML type String.

16.5 Data Members, set Functions and get Functions

Variables declared in a function definition’s body are known as local variables and can be used only from the line of their declaration in the function to closing right brace (}) of the block in which they’re declared. A local variable must be declared before it can be used in a function. A local variable cannot be accessed outside the function in which it’s declared. When a function terminates, the values of its local variables are lost. Recall from Section 16.2 that an object has attributes that are carried with it as it’s used in a program. Such attributes exist throughout the life of the object.

A class normally consists of one or more member functions that manipulate the attributes that belong to a particular object of the class. Attributes are represented as variables in a class definition. Such variables are called data members and are declared inside a class definition but outside the bodies of the class’s member-function definitions. Each object of a class maintains its own copy of its attributes in memory. The example in this section demonstrates a GradeBook class that contains a courseName data member to represent a particular GradeBook object’s course name.
Chapter 16  Introduction to Classes and Objects

GradeBook Class with a Data Member, a set Function and a get Function

In our next example, class GradeBook (Fig. 16.5) maintains the course name as a data member so that it can be used or modified at any time during a program’s execution. The class contains member functions setCourseName, getCourseName and displayMessage. Member function setCourseName stores a course name in a GradeBook data member. Member function getCourseName obtains the course name from that data member. Member function displayMessage—which now specifies no parameters—still displays a welcome message that includes the course name. However, as you’ll see, the function now obtains the course name by calling another function in the same class—getCourseName.

Good Programming Practice 16.3

Place a blank line between member-function definitions to enhance program readability.

A typical instructor teaches multiple courses, each with its own course name. Line 34 declares that courseName is a variable of type string. Because the variable is declared in the class definition (lines 10–35) but outside the bodies of the class’s member-function definitions (lines 14–17, 20–23 and 26–32), the variable is a data member. Every instance (i.e., object) of class GradeBook contains one copy of each of the class’s data members—if there are two GradeBook objects, each has its own copy of courseName (one per object), as you’ll see in the example of Fig. 16.7. A benefit of making courseName a data member is that all the member functions of the class (in this case, class GradeBook) can manipulate any data members that appear in the class definition (in this case, courseName).

Fig. 16.5  |  Defining and testing class GradeBook with a data member and set and get functions.  
(Part 1 of 2.)
Access Specifiers `public` and `private`

Most data-member declarations appear after the access-specifier label `private:` (line 33). Like `public`, keyword `private` is an access specifier. Variables or functions declared after access specifier `private` (and before the next access specifier) are accessible only to member functions of the class for which they’re declared. Thus, data member `courseName` can be used only in member functions `setCourseName`, `getCourseName` and `displayMessage` of (every object of) class `GradeBook`. Data member `courseName`, because it’s `private`, cannot be accessed by functions outside the class (such as `main`) or by member functions of other classes in the program. Attempting to access data member `courseName` in one of these pro-

```cpp
private:

string courseName; // course name for this GradeBook

// function that displays a welcome message
void displayMessage()
{
    // this statement calls getCourseName to get the
    // name of the course this GradeBook represents
    cout << "Welcome to the grade book for\n" << getCourseName() << "!" << endl;
}

// function main begins program execution
int main()
{
    string nameOfCourse; // string of characters to store the course name
    GradeBook myGradeBook; // create a GradeBook object named myGradeBook

    // display initial value of courseName
    cout << "Initial course name is: " << myGradeBook.getCourseName() << endl;

    // prompt for, input and set course name
    cout << "Please enter the course name:" << endl;
    getline( cin, nameOfCourse ); // read a course name with blanks
    myGradeBook.setCourseName( nameOfCourse ); // set the course name

    cout << endl; // outputs a blank line
    myGradeBook.displayMessage(); // display message with new course name
}
```

---

**Fig. 16.5** Defining and testing class `GradeBook` with a data member and `set` and `get` functions. (Part 2 of 2.)
gram locations with an expression such as myGradeBook.courseName would result in a compilation error containing a message similar to:

```
cannot access private member declared in class 'GradeBook'
```

**Software Engineering Observation 16.1**
Generally, data members should be declared private and member functions should be declared public. (We’ll see that it’s appropriate to declare certain member functions private, if they’re to be accessed only by other member functions of the class.)

**Common Programming Error 16.6**
An attempt by a function, which is not a member of a particular class (or a friend of that class, as we’ll see in Chapter 18, Classes: A Deeper Look, Part 2), to access a private member of that class is a compilation error.

The default access for class members is private so all members after the class header and before the first access specifier are private. The access specifiers public and private may be repeated, but this is unnecessary and can be confusing.

**Good Programming Practice 16.4**
Despite the fact that the public and private access specifiers may be repeated and intermixed, list all the public members of a class first in one group then list all the private members in another group. This focuses the programmer’s attention on the class’s public interface, rather than on the class’s implementation.

**Good Programming Practice 16.5**
If you choose to list the private members first in a class definition, explicitly use the private access specifier despite the fact that private is assumed by default. This improves program clarity.

Declaring data members with access specifier private is known as data hiding. When a program creates (instantiates) a GradeBook object, data member courseName is encapsulated (hidden) in the object and can be accessed only by member functions of the object’s class. In class GradeBook, member functions setCourseName and getCourseName manipulate the data member courseName directly (and displayMessage could do so if necessary).

**Software Engineering Observation 16.2**
You’ll learn in Chapter 18 that functions and classes declared by a class to be “friends” can access the private members of the class.

**Error-Prevention Tip 16.1**
Making the data members of a class private and the member functions of the class public facilitates debugging because problems with data manipulations are localized to either the class’s member functions or the friends of the class.

**Member Functions setCourseName and getCourseName**
Member function setCourseName (defined in lines 14–17) does not return any data when it completes its task, so its return type is void. The member function receives one param-
eter—name—which represents the course name that will be passed to it as an argument (as we’ll see in line 50 of main). Line 16 assigns name to data member courseName. In this example, setCourseName does not attempt to validate the course name—i.e., the function does not check that the course name adheres to any particular format or follows any other rules regarding what a “valid” course name looks like. Suppose, for instance, that a university can print student transcripts containing course names of only 25 characters or fewer. In this case, we might want class GradeBook to ensure that its data member courseName never contains more than 25 characters. We discuss basic validation techniques in Section 16.9.

Member function getCourseName (defined in lines 20–23) returns a particular GradeBook object’s courseName. The member function has an empty parameter list, so it does not require additional data to perform its task. The function specifies that it returns a string. When a function that specifies a return type other than void is called and completes its task, the function uses a return statement (as in line 22) to return a result to its calling function. For example, when you go to an automated teller machine (ATM) and request your account balance, you expect the ATM to give you back a value that represents your balance. Similarly, when a statement calls member function getCourseName on a GradeBook object, the statement expects to receive the GradeBook’s course name (in this case, a string, as specified by the function’s return type). If you have a function square that returns the square of its argument, the statement

\[
result = \text{square}(2);
\]

returns 4 from function square and assigns to variable result the value 4. If you have a function maximum that returns the largest of three integer arguments, the statement

\[
biggest = \text{maximum}(27, 114, 51);
\]

returns 114 from function maximum and assigns to variable biggest the value 114.

**Common Programming Error 16.7**

Forgetting to return a value from a function that is supposed to return a value is a compilation error.

The statements in lines 16 and 22 each use variable courseName (line 34) even though it was not declared in any of the member functions. We can use courseName in the member functions of class GradeBook because courseName is a data member of the class. So member function getCourseName could be defined before member function setCourseName.

**Member Function displayMessage**

Member function displayMessage (lines 26–32) does not return any data when it completes its task, so its return type is void. The function does not receive parameters, so its parameter list is empty. Lines 30–31 output a welcome message that includes the value of data member courseName. Line 30 calls member function getCourseName to obtain the value of courseName. Member function displayMessage could also access data member courseName directly, just as member functions setCourseName and getCourseName do. We explain shortly why we choose to call member function getCourseName to obtain the value of courseName.
Testing Class GradeBook
The main function (lines 38–54) creates one object of class GradeBook and uses each of its member functions. Line 41 creates a GradeBook object named myGradeBook. Lines 44–45 display the initial course name by calling the object's getCourseName member function. The first line of the output does not show a course name, because the object's courseName data member (i.e., a string) is initially empty—by default, the initial value of a string is the so-called empty string, i.e., a string that does not contain any characters. Nothing appears on the screen when an empty string is displayed.

Line 48 prompts the user to enter a course name. Local string variable nameOfCourse (declared in line 40) is set to the course name entered by the user, which is obtained by the call to the getline function (line 49). Line 50 calls object myGradeBook's setCourseName member function and supplies nameOfCourse as the function's argument. When the function is called, the argument's value is copied to parameter name (line 14) of member function setCourseName. Then the parameter's value is assigned to data member courseName (line 16). Line 52 skips a line; then line 53 calls object myGradeBook's displayMessage member function to display the welcome message containing the course name.

Software Engineering with Set and Get Functions
A class's private data members can be manipulated only by member functions of that class (and by "friends" of the class, as we'll see in Chapter 18). So a client of an object—that is, any class or function that calls the object's member functions from outside the object—calls the class's public member functions to request the class's services for particular objects of the class. This is why the statements in function main call member functions setCourseName, getCourseName and displayMessage on a GradeBook object. Classes often provide public member functions to allow clients of the class to set (i.e., assign values to) or get (i.e., obtain the values of) private data members. These member function names need not begin with set or get, but this naming convention is common. In this example, the member function that sets the courseName data member is called setCourseName, and the member function that gets the value of the courseName data member is called getCourseName. Set functions are also sometimes called mutators (because they mutate, or change, values), and get functions are also sometimes called accessorrs (because they access values).

Recall that declaring data members with access specifier private enforces data hiding. Providing public set and get functions allows clients of a class to access the hidden data, but only indirectly. The client knows that it's attempting to modify or obtain an object's data, but the client does not know how the object performs these operations. In some cases, a class may internally represent a piece of data one way, but expose that data to clients in a different way. For example, suppose a Clock class represents the time of day as a private int data member time that stores the number of seconds since midnight. However, when a client calls a Clock object's getTime member function, the object could return the time with hours, minutes and seconds in a string in the format "HH:MM:SS". Similarly, suppose the Clock class provides a set function named setTime that takes a string parameter in the "HH:MM:SS" format. Using string class capabilities, the setTime function could convert this string to a number of seconds, which the function stores in its private data member. The set function could also check that the value it receives represents a valid time (e.g., "12:30:45" is valid but "42:85:70" is not). The set and get func-
Data Members, set Functions and get Functions

The set and get functions of a class also should be used by other member functions within the class to manipulate the class’s private data, although these member functions can access the private data directly. In Fig. 16.5, member functions setCourseName and getCourseName are public member functions, so they’re accessible to clients of the class, as well as to the class itself. Member function displayMessage calls member function getCourseName to obtain the value of data member courseName for display purposes, even though displayMessage can access courseName directly—accessing a data member via its get function creates a better, more robust class (i.e., a class that is easier to maintain and less likely to stop working). If we decide to change the data member courseName in some way, the displayMessage definition will not require modification—only the bodies of the get and set functions that directly manipulate the data member will need to change. For example, suppose we want to represent the course name as two separate data members—courseNumber (e.g., "CS101") and courseTitle (e.g., "Introduction to C++ Programming"). Member function displayMessage can still issue a single call to member function getCourseName to obtain the full course name to display as part of the welcome message. In this case, getCourseName would need to build and return a string containing the courseNumber followed by the courseTitle. Member function displayMessage would continue to display the complete course title “CS101 Introduction to C++ Programming,” because it’s unaffected by the change to the class’s data members. The benefits of calling a set function from another member function of a class will become clear when we discuss validation in Section 16.9.

**Good Programming Practice 16.6**
Always try to localize the effects of changes to a class’s data members by accessing and manipulating the data members through their get and set functions. Changes to the name of a data member or the data type used to store a data member then affect only the corresponding get and set functions, but not the callers of those functions.

**Software Engineering Observation 16.3**
Write programs that are understandable and easy to maintain. Change is the rule rather than the exception. You should anticipate that your code will be modified.

**Software Engineering Observation 16.4**
Provide set or get functions for each private data item only when appropriate. Services useful to the client should typically be provided in the class’s public interface.

**GradeBook’s UML Class Diagram with a Data Member and set and get Functions**
Figure 16.6 contains an updated UML class diagram for the version of class GradeBook in Fig. 16.5. This diagram models GradeBook’s data member courseName as an attribute in the middle compartment. The UML represents data members as attributes by listing the attribute name, followed by a colon and the attribute type. The UML type of attribute courseName is String, which corresponds to string in C++. Data member courseName is private in C++, so the class diagram lists a minus sign (−) in front of the corresponding
attribute’s name. The minus sign in the UML is equivalent to the private access specifier in C++. Class GradeBook contains three public member functions, so the class diagram lists three operations in the third compartment. Operation setCourseName has a String parameter called name. The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name. Member function getCourseName of class GradeBook has a string return type in C++, so the class diagram shows a String return type in the UML. Operations setCourseName and displayMessage do not return values (i.e., they return void), so the UML class diagram does not specify a return type after the parentheses of these operations.

![Fig. 16.6 | UML class diagram for class GradeBook with a private courseName attribute and public operations setCourseName, getCourseName and displayMessage.](image)

### 16.6 Initializing Objects with Constructors

As mentioned in Section 16.5, when an object of class GradeBook (Fig. 16.5) is created, its data member courseName is initialized to the empty string by default. What if you want to provide a course name when you create a GradeBook object? Each class you declare can provide a constructor that can be used to initialize an object of the class when the object is created. A constructor is a special member function that must be defined with the same name as the class, so that the compiler can distinguish it from the class’s other member functions. An important difference between constructors and other functions is that constructors cannot return values, so they cannot specify a return type (not even void). Normally, constructors are declared public.

C++ requires a constructor call for each object that is created, which helps ensure that each object is initialized before it’s used in a program. The constructor call occurs implicitly when the object is created. If a class does not explicitly include a constructor, the compiler provides a default constructor—that is, a constructor with no parameters. For example, when line 41 of Fig. 16.5 creates a GradeBook object, the default constructor is called. The default constructor provided by the compiler creates a GradeBook object without giving any initial values to the object’s fundamental type data members. [Note: For data members that are objects of other classes, the default constructor implicitly calls each data member’s default constructor to ensure that the data member is initialized properly. This is why the string data member courseName (in Fig. 16.5) was initialized to the empty string—the default constructor for class string sets the string’s value to the empty string. You’ll learn more about initializing data members that are objects of other classes in Section 18.3.]

In the example of Fig. 16.7, we specify a course name for a GradeBook object when the object is created (e.g., line 46). In this case, the argument "CS101 Introduction to C++
Programming" is passed to the GradeBook object's constructor (lines 14–17) and used to initialize the courseName. Figure 16.7 defines a modified GradeBook class containing a constructor with a string parameter that receives the initial course name.

```cpp
1 // Fig. 16.7: fig16_07.cpp
2 // Instantiating multiple objects of the GradeBook class and using
3 // the GradeBook constructor to specify the course name
4 // when each GradeBook object is created.
5 #include <iostream>
6 #include <string> // program uses C++ standard string class
7 using namespace std;
8
9 // GradeBook class definition
10 class GradeBook
11 {
12 public:
13     // function to set the course name
14     void setCourseName( string name )
15     {
16         courseName = name; // store the course name in the object
17     } // end function setCourseName
18
19     // function to get the course name
20     string getCourseName()
21     {
22         return courseName; // return object's courseName
23     } // end function getCourseName
24
25     // display a welcome message to the GradeBook user
26     void displayMessage()
27     {
28         // call getCourseName to get the courseName
29         cout << "Welcome to the grade book for\n" << getCourseName()
30             << "!" << endl;
31     } // end function displayMessage
32
33     private:
34         string courseName; // course name for this GradeBook
35     }; // end class GradeBook
36
37 // function main begins program execution
38 int main()
39 {
40     // create two GradeBook objects
41     GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
42     GradeBook gradeBook2( "CS102 Data Structures in C++" );
```
Defining a Constructor

Lines 14–17 of Fig. 16.7 define a constructor for class GradeBook. Notice that the constructor has the same name as its class, GradeBook. A constructor specifies in its parameter list the data it requires to perform its task. When you create a new object, you place this data in the parentheses that follow the object name (as we did in lines 46–47). Line 14 indicates that class GradeBook’s constructor has a string parameter called name. Line 14 does not specify a return type, because constructors cannot return values (or even void).

Line 16 in the constructor’s body passes the constructor’s parameter name to member function setCourseName (lines 20–23), which simply assigns the value of its parameter to data member courseName. You might be wondering why we bother making the call to setCourseName in line 16—the constructor certainly could perform the assignment courseName = name. In Section 16.9, we modify setCourseName to perform validation (ensuring that, in this case, the courseName is 25 or fewer characters in length). At that point the benefits of calling setCourseName from the constructor will become clear. Both the constructor (line 14) and the setCourseName function (line 20) use a parameter called name. You can use the same parameter names in different functions because the parameters are local to each function; they do not interfere with one another.

Testing Class GradeBook

Lines 43–53 of Fig. 16.7 define the main function that tests class GradeBook and demonstrates initializing GradeBook objects using a constructor. Line 46 creates and initializes a GradeBook object called gradeBook1. When this line executes, the GradeBook constructor (lines 14–17) is called (implicitly by C++) with the argument "CS101 Introduction to C++ Programming" to initialize gradeBook1’s course name. Line 47 repeats this process for the GradeBook object called gradeBook2, this time passing the argument "CS102 Data Structures in C++" to initialize gradeBook2’s course name. Lines 50–51 use each object’s getCourseName member function to obtain the course names and show that they were indeed initialized when the objects were created. The output confirms that each GradeBook object maintains its own copy of data member courseName.

Two Ways to Provide a Default Constructor for a Class

Any constructor that takes no arguments is called a default constructor. A class gets a default constructor in one of two ways:
1. The compiler implicitly creates a default constructor in a class that does not define a constructor. Such a constructor does not initialize the class’s data members, but does call the default constructor for each data member that is an object of another class. An uninitialized variable typically contains a “garbage” value.

2. You explicitly define a constructor that takes no arguments. Such a default constructor will call the default constructor for each data member that is an object of another class and will perform additional initialization specified by you.

If you define a constructor with arguments, C++ will not implicitly create a default constructor for that class. For each version of class GradeBook in Fig. 16.1, Fig. 16.3 and Fig. 16.5 the compiler implicitly defined a default constructor.

**Error-Prevention Tip 16.2**
Unless no initialization of your class’s data members is necessary (almost never), provide a constructor to ensure that your class’s data members are initialized with meaningful values when each new object of your class is created.

**Software Engineering Observation 16.5**
Data members can be initialized in a constructor, or their values may be set later after the object is created. However, it’s a good software engineering practice to ensure that an object is fully initialized before the client code invokes the object’s member functions. You should not rely on the client code to ensure that an object gets initialized properly.

**Adding the Constructor to Class GradeBook’s UML Class Diagram**
The UML class diagram of Fig. 16.8 models class GradeBook of Fig. 16.7, which has a constructor with a name parameter of type string (represented by type String in the UML). Like operations, the UML models constructors in the third compartment of a class in a class diagram. To distinguish a constructor from a class’s operations, the UML places the word “constructor” between guillemets (« and ») before the constructor’s name. It’s customary to list the class’s constructor before other operations in the third compartment.

![UML class diagram indicating that class GradeBook has a constructor with a name parameter of UML type String.](image)

**16.7 Placing a Class in a Separate File for Reusability**
One of the benefits of creating class definitions is that, when packaged properly, our classes can be reused by programmers—potentially worldwide. For example, we can reuse C++
Standard Library type string in any C++ program by including the header file `<string>` (and, as we’ll see, by being able to link to the library’s object code).

Programmers who wish to use our `GradeBook` class cannot simply include the file from Fig. 16.7 in another program. As you know, function `main` begins the execution of every program, and every program must have exactly one `main` function. If other programmers include the code from Fig. 16.7, they get extra baggage—our `main` function—and their programs will then have two `main` functions. Attempting to compile a program with two `main` functions in Microsoft Visual C++ produces an error such as

```
error C2084: function 'int main(void)' already has a body
```

when the compiler tries to compile the second `main` function it encounters. Similarly, the GNU C++ compiler produces the error

```
redefinition of 'int main()'
```

These errors indicate that a program already has a `main` function. So, placing `main` in the same file with a class definition prevents that class from being reused by other programs. In this section, we demonstrate how to make class `GradeBook` reusable by separating it into another file from the `main` function.

**Header Files**

Each of the previous examples in the chapter consists of a single `.cpp` file, also known as a source-code file, that contains a `GradeBook` class definition and a `main` function. When building an object-oriented C++ program, it’s customary to define reusable source code (such as a class) in a file that by convention has a `.h` filename extension—known as a header file. Programs use `#include` preprocessor directives to include header files and take advantage of reusable software components, such as type `string` provided in the C++ Standard Library and user-defined types like class `GradeBook`.

Our next example separates the code from Fig. 16.7 into two files—`GradeBook.h` (Fig. 16.9) and `fig16_10.cpp` (Fig. 16.10). As you look at the header file in Fig. 16.9, notice that it contains only the `GradeBook` class definition (lines 8–38), the appropriate header files and a `using` declaration. The `main` function that uses class `GradeBook` is defined in the source-code file `fig16_10.cpp` (Fig. 16.10) in lines 8–18. To help you prepare for the larger programs you’ll encounter later in this book and in industry, we often use a separate source-code file containing function `main` to test our classes (this is called a driver program). You’ll soon learn how a source-code file with `main` can use the class definition found in a header file to create objects of a class.

```cpp
1 // Fig. 16.9: GradeBook.h
2 // GradeBook class definition in a separate file from main.
3 #include <iostream>
4 #include <string> // class GradeBook uses C++ standard string class
5 using namespace std;
6
7 // GradeBook class definition
8 class GradeBook
9 {
```

`Fig. 16.9` | `GradeBook` class definition in a separate file from `main`. (Part 1 of 2.)
public:
   // constructor initializes courseName with string supplied as argument
   GradeBook( string name )
   {
      setCourseName( name ); // call set function to initialize courseName
   } // end GradeBook constructor

   // function to set the course name
   void setCourseName( string name )
   {
      courseName = name; // store the course name in the object
   } // end function setCourseName

   // function to get the course name
   string getCourseName()
   {
      return courseName; // return object's courseName
   } // end function getCourseName

   // display a welcome message to the GradeBook user
   void displayMessage()
   {
      // call getCourseName to get the courseName
      cout << "Welcome to the grade book for\n" << getCourseName() << "!
   } // end function displayMessage

private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook

Fig. 16.9 | GradeBook class definition in a separate file from main. (Part 2 of 2.)

// Fig. 16.10: fig16_10.cpp
// Including class GradeBook from file GradeBook.h for use in main.
#include <iostream>
#include "GradeBook.h" // include definition of class GradeBook
using namespace std;

// function main begins program execution
int main()
{
   // create two GradeBook objects
   GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
   GradeBook gradeBook2( "CS102 Data Structures in C++" );

   // display initial value of courseName for each GradeBook
   cout << "gradeBook1 created for course: " << gradeBook1.getCourseName() << endl;
   cout << "gradeBook2 created for course: " << gradeBook2.getCourseName() << endl;
} // end main

Fig. 16.10 | Including class GradeBook from file GradeBook.h for use in main. (Part 1 of 2.)
A header file such as GradeBook.h (Fig. 16.9) cannot be used to begin program execution, because it does not contain a main function. If you try to compile and link GradeBook.h by itself to create an executable application, Microsoft Visual C++ 2008 produces the linker error message:

error LNK2001: unresolved external symbol _mainCRTStartup

To compile and link with GNU C++ on Linux, you must first include the header file in a .cpp source-code file, then GNU C++ produces a linker error message containing:

undefined reference to 'main'

This error indicates that the linker could not locate the program’s main function. To test class GradeBook (defined in Fig. 16.9), you must write a separate source-code file containing a main function (such as Fig. 16.10) that instantiates and uses objects of the class.

The compiler does not know what a GradeBook is because it’s a user-defined type. In fact, the compiler doesn’t even know the classes in the C++ Standard Library. To help it understand how to use a class, we must explicitly provide the compiler with the class’s definition—that’s why, for example, to use type string, a program must include the <string> header file. This enables the compiler to determine the amount of memory that it must reserve for each object of the class and ensure that a program calls the class’s member functions correctly.

To create GradeBook objects gradeBook1 and gradeBook2 in lines 11–12 of Fig. 16.10, the compiler must know the size of a GradeBook object. While objects conceptually contain data members and member functions, C++ objects contain only data. The compiler creates only one copy of the class’s member functions and shares that copy among all the class’s objects. Each object, of course, needs its own copy of the class’s data members, because their contents can vary among objects (such as two different BankAccount objects having two different balance data members). The member-function code, however, is not modifiable, so it can be shared among all objects of the class. Therefore, the size of an object depends on the amount of memory required to store the class’s data members. By including GradeBook.h in line 4, we give the compiler access to the information it needs (Fig. 16.9, line 37) to determine the size of a GradeBook object and to determine whether objects of the class are used correctly (in lines 11–12 and 15–16 of Fig. 16.10).

Line 4 instructs the C++ preprocessor to replace the directive with a copy of the contents of GradeBook.h (i.e., the GradeBook class definition) before the program is compiled. When the source-code file fig16_10.cpp is compiled, it now contains the GradeBook class definition (because of the #include), and the compiler is able to determine how to create GradeBook objects and see that their member functions are called correctly. Now that the class definition is in a header file (without a main function), we can include that header in any program that needs to reuse our GradeBook class.
How Header Files Are Located

Notice that the name of the GradeBook.h header file in line 4 of Fig. 16.10 is enclosed in quotes (""") rather than angle brackets (<>). Normally, a program’s source-code files and user-defined header files are placed in the same directory. When the preprocessor encounters a header file name in quotes, it attempts to locate the header file in the same directory as the file in which the #include directive appears. If the preprocessor cannot find the header file in that directory, it searches for it in the same location(s) as the C++ Standard Library header files. When the preprocessor encounters a header file name in angle brackets (e.g., <iostream>), it assumes that the header is part of the C++ Standard Library and does not look in the directory of the program that is being preprocessed.

Additional Software Engineering Issues

Now that class GradeBook is defined in a header file, the class is reusable. Unfortunately, placing a class definition in a header file as in Fig. 16.9 still reveals the entire implementation of the class to the class’s clients—GradeBook.h is simply a text file that anyone can open and read. Conventional software engineering wisdom says that to use an object of a class, the client code needs to know only what member functions to call, what arguments to provide to each member function and what return type to expect from each member function. The client code does not need to know how those functions are implemented.

If client code does know how a class is implemented, the client-code programmer might write client code based on the class’s implementation details. Ideally, if that implementation changes, the class’s clients should not have to change. Hiding the class’s implementation details makes it easier to change the class’s implementation while minimizing, and hopefully eliminating, changes to client code.

In Section 16.8, we show how to break up the GradeBook class into two files so that

1. the class is reusable,
2. the clients of the class know what member functions the class provides, how to call them and what return types to expect, and
3. the clients do not know how the class’s member functions are implemented.

16.8 Separating Interface from Implementation

In the preceding section, we showed how to promote software reusability by separating a class definition from the client code (e.g., function main) that uses the class. We now introduce another fundamental principle of good software engineering—separating interface from implementation.

Interface of a Class

Interfaces define and standardize the ways in which things such as people and systems interact with one another. For example, a radio’s controls serve as an interface between the
Chapter 16  Introduction to Classes and Objects

radio’s users and its internal components. The controls allow users to perform a limited set of operations (such as changing the station, adjusting the volume, and choosing between AM and FM stations). Various radios may implement these operations differently—some provide push buttons, some provide dials and some support voice commands. The interface specifies what operations a radio permits users to perform but does not specify how the operations are implemented inside the radio.

Similarly, the interface of a class describes what services a class’s clients can use and how to request those services, but not how the class carries out the services. A class’s public interface consists of the class’s public member functions (also known as the class’s public services). For example, class GradeBook’s interface (Fig. 16.9) contains a constructor and member functions setCourseName, getCourseName and displayMessage. GradeBook’s clients (e.g., main in Fig. 16.10) use these functions to request the class’s services. As you’ll soon see, you can specify a class’s interface by writing a class definition that lists only the member-function names, return types and parameter types.

Separating the Interface from the Implementation
In our prior examples, each class definition contained the complete definitions of the class’s public member functions and the declarations of its private data members. However, it’s better software engineering to define member functions outside the class definition, so that their implementation details can be hidden from the client code. This practice ensures that you do not write client code that depends on the class’s implementation details. If you were to do so, the client code would be more likely to “break” if the class’s implementation changed.

The program of Figs. 16.11–16.13 separates class GradeBook’s interface from its implementation by splitting the class definition of Fig. 16.9 into two files—the header file GradeBook.h (Fig. 16.11) in which class GradeBook is defined, and the source-code file GradeBook.cpp (Fig. 16.12) in which GradeBook’s member functions are defined. By convention, member-function definitions are placed in a source-code file of the same base name (e.g., GradeBook) as the class’s header file but with a .cpp filename extension. The source-code file fig16_13.cpp (Fig. 16.13) defines function main (the client code). The code and output of Fig. 16.13 are identical to that of Fig. 16.10. Figure 16.14 shows how this three-file program is compiled from the perspectives of the GradeBook class programmer and the client-code programmer—we’ll explain this figure in detail.

GradeBook.h: Defining a Class’s Interface with Function Prototypes
Header file GradeBook.h (Fig. 16.11) contains another version of GradeBook’s class definition (lines 9–18). This version is similar to the one in Fig. 16.9, but the function definitions in Fig. 16.9 are replaced here with function prototypes (lines 12–15) that describe the class’s public interface without revealing the class’s member-function implementations. A function prototype is a declaration of a function that tells the compiler the function’s name, its return type and the types of its parameters. Also, the header file still specifies the class’s private data member (line 17) as well. Again, the compiler must know the data members of the class to determine how much memory to reserve for each object of the class. Including the header file GradeBook.h in the client code (line 5 of Fig. 16.13) provides the compiler with the information it needs to ensure that the client code calls the member functions of class GradeBook correctly.
The function prototype in line 12 (Fig. 16.11) indicates that the constructor requires one string parameter. Recall that constructors do not have return types, so no return type appears in the function prototype. Member function setCourseName’s function prototype indicates that setCourseName requires a string parameter and does not return a value (i.e., its return type is void). Member function getCourseName’s function prototype indicates that the function does not require parameters and returns a string. Finally, member function displayMessage’s function prototype (line 15) specifies that displayMessage does not require parameters and does not return a value. These function prototypes are the same as the corresponding function headers in Fig. 16.9, except that the parameter names (which are optional in prototypes) are not included and each function prototype must end with a semicolon.

Common Programming Error 16.8
Forgetting the semicolon at the end of a function prototype is a syntax error.

Good Programming Practice 16.7
Although parameter names in function prototypes are optional (they’re ignored by the compiler), many programmers use these names for documentation purposes.

Error-Prevention Tip 16.4
Parameter names in a function prototype (which, again, are ignored by the compiler) can be misleading if the names used do not match those used in the function definition. For this reason, many programmers create function prototypes by copying the first line of the corresponding function definitions (when the source code for the functions is available), then appending a semicolon to the end of each prototype.

Fig. 16.11  GradeBook class definition containing function prototypes that specify the interface of the class.
Chapter 16  Introduction to Classes and Objects

GradeBook.cpp: Defining Member Functions in a Separate Source-Code File

Source-code file GradeBook.cpp (Fig. 16.12) defines class GradeBook’s member functions, which were declared in lines 12–15 of Fig. 16.11. The definitions appear in lines 9–32 and are nearly identical to the member-function definitions in lines 12–35 of Fig. 16.9.

```
1  // Fig. 16.12: GradeBook.cpp
2  // GradeBook member-function definitions. This file contains
3  // implementations of the member functions prototyped in GradeBook.h.
4  #include <iostream>
5  #include "GradeBook.h"  // include definition of class GradeBook
6  using namespace std;
7  
8  // constructor initializes courseName with string supplied as argument
9  GradeBook::GradeBook( string name )
10  {
11    setCourseName( name ); // call set function to initialize courseName
12  } // end GradeBook constructor
13  
14  // function to set the course name
15  void GradeBook::setCourseName( string name )
16  {
17    courseName = name; // store the course name in the object
18  } // end function setCourseName
19  
20  // function to get the course name
21  string GradeBook::getCourseName()
22  {
23    return courseName; // return object's courseName
24  } // end function getCourseName
25  
26  // display a welcome message to the GradeBook user
27  void GradeBook::displayMessage()
28  {
29    // call getCourseName to get the courseName
30    cout << "Welcome to the grade book for\n" << getCourseName()
31      << "!
" << endl;
32  } // end function displayMessage
```

Fig. 16.12  GradeBook member-function definitions represent the implementation of class GradeBook.

Notice that each member-function name in the function headers (lines 9, 15, 21 and 27) is preceded by the class name and ::, which is known as the binary scope resolution operator. This “ties” each member function to the (now separate) GradeBook class definition (Fig. 16.11), which declares the class’s member functions and data members. Without “GradeBook::” preceding each function name, these functions would not be recognized by the compiler as member functions of class GradeBook—the compiler would consider them “free” or “loose” functions, like main. These are also called global functions. Such functions cannot access GradeBook’s private data or call the class’s member functions, without specifying an object. So, the compiler would not be able to compile these functions. For example, lines 17 and 23 that access variable courseName would cause compilation errors.
because `courseName` is not declared as a local variable in each function—the compiler would not know that `courseName` is already declared as a data member of class `GradeBook`.

**Common Programming Error 16.9**
When defining a class’s member functions outside that class, omitting the class name and binary scope resolution operator (::) preceding the function names causes compilation errors.

To indicate that the member functions in `GradeBook.cpp` are part of class `GradeBook`, we must first include the `GradeBook.h` header file (line 5 of Fig. 16.12). This allows us to access the class name `GradeBook` in the `GradeBook.cpp` file. When compiling `GradeBook.cpp`, the compiler uses the information in `GradeBook.h` to ensure that

1. the first line of each member function (lines 9, 15, 21 and 27) matches its prototype in the `GradeBook.h` file—for example, the compiler ensures that `getCourseName` accepts no parameters and returns a string, and that

2. each member function knows about the class’s data members and other member functions—for example, lines 17 and 23 can access variable `courseName` because it’s declared in `GradeBook.h` as a data member of class `GradeBook`, and lines 11 and 30 can call functions `setCourseName` and `getCourseName`, respectively, because each is declared as a member function of the class in `GradeBook.h` (and because these calls conform with the corresponding prototypes).

**Testing Class GradeBook**
Figure 16.13 performs the same `GradeBook` object manipulations as Fig. 16.10. Separating `GradeBook`’s interface from the implementation of its member functions does not affect the way that this client code uses the class. It affects only how the program is compiled and linked, which we discuss in detail shortly.

```cpp
// Fig. 16.13: fig16_13.cpp
// GradeBook class demonstration after separating its interface from its implementation.
#include <iostream>
#include "GradeBook.h" // include definition of class GradeBook
using namespace std;

// function main begins program execution
int main()
{
    // create two GradeBook objects
    GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
    GradeBook gradeBook2( "CS102 Data Structures in C++" );

    // display initial value of courseName for each GradeBook
    cout << "gradeBook1 created for course: " << gradeBook1.getCourseName() << "gradeBook2 created for course: " << gradeBook2.getCourseName() << endl;
} // end main
```

Fig. 16.13 | GradeBook class demonstration after separating its interface from its implementation. (Part 1 of 2.)
As in Fig. 16.10, line 5 of Fig. 16.13 includes the `GradeBook.h` header file so that the compiler can ensure that `GradeBook` objects are created and manipulated correctly in the client code. Before executing this program, the source-code files in Fig. 16.12 and Fig. 16.13 must both be compiled, then linked together—that is, the member-function calls in the client code need to be tied to the implementations of the class’s member functions—a job performed by the linker.

The Compilation and Linking Process
The diagram in Fig. 16.14 shows the compilation and linking process that results in an executable `GradeBook` application that can be used by instructors. Often a class’s interface and implementation will be created and compiled by one programmer and used by a separate programmer who implements the client code that uses the class. So, the diagram shows what’s required by both the class-implementation programmer and the client-code programmer. The dashed lines in the diagram show the pieces required by the class-implementation programmer, the client-code programmer and the `GradeBook` application user, respectively. [Note: Figure 16.14 is not a UML diagram.]

A class-implementation programmer responsible for creating a reusable `GradeBook` class creates the header file `GradeBook.h` and the source-code file `GradeBook.cpp` that `#include`s the header file, then compiles the source-code file to create `GradeBook`’s object code. To hide the class’s member-function implementation details, the class-implementation programmer would provide the client-code programmer with the header file `GradeBook.h` (which specifies the class’s interface and data members) and the `GradeBook` object code (i.e., the machine-language instructions that represent `GradeBook`’s member functions). The client-code programmer is not given `GradeBook.cpp`, so the client remains unaware of how `GradeBook`’s member functions are implemented.

The client code needs to know only `GradeBook`’s interface to use the class and must be able to link its object code. Since the interface of the class is part of the class definition in the `GradeBook.h` header file, the client-code programmer must have access to this file and must `#include` it in the client’s source-code file. When the client code is compiled, the compiler uses the class definition in `GradeBook.h` to ensure that the `main` function creates and manipulates objects of class `GradeBook` correctly.

To create the executable `GradeBook` application, the last step is to link

1. the object code for the `main` function (i.e., the client code),
2. the object code for class `GradeBook`’s member-function implementations and
3. the C++ Standard Library object code for the C++ classes (e.g., `string`) used by the class-implementation programmer and the client-code programmer.

The linker’s output is the executable `GradeBook` application that instructors can use to manage their students’ grades. Compilers and IDEs typically invoke the linker for you after compiling your code.
For further information on compiling multiple-source-file programs, see your compiler's documentation. We provide links to various C++ compilers in our C++ Resource Center at www.deitel.com/cplusplus/.

16.9 Validating Data with set Functions

In Section 16.5, we introduced set functions for allowing clients of a class to modify the value of a private data member. In Fig. 16.5, class GradeBook defines member function
setCourseName to simply assign a value received in its parameter name to data member courseName. This member function does not ensure that the course name adheres to any particular format or follows any other rules regarding what a “valid” course name looks like. As we stated earlier, suppose that a university can print student transcripts containing course names of only 25 characters or less. If the university uses a system containing GradeBook objects to generate the transcripts, we might want class GradeBook to ensure that its data member courseName never contains more than 25 characters. The program of Figs. 16.15–16.17 enhances class GradeBook’s member function setCourseName to perform this validation (also known as validity checking).

**GradeBook Class Definition**

Notice that GradeBook’s class definition (Fig. 16.15)—and hence, its interface—is identical to that of Fig. 16.11. Since the interface remains unchanged, clients of this class need not be changed when the definition of member function setCourseName is modified. This enables clients to take advantage of the improved GradeBook class simply by linking the client code to the updated GradeBook’s object code.

```cpp
// Fig. 16.15: GradeBook.h
// GradeBook class definition presents the public interface of
// the class. Member-function definitions appear in GradeBook.cpp.
#include <string> // program uses C++ standard string class
using namespace std;

class GradeBook
{
public:
    GradeBook( string ); // constructor that initializes a GradeBook object
    void setCourseName( string ); // function that sets the course name
    string getCourseName(); // function that gets the course name
    void displayMessage(); // function that displays a welcome message
private:
    string courseName; // course name for this GradeBook
}; // end class GradeBook
```

**Fig. 16.15 | GradeBook class definition.**

**Validating the Course Name with GradeBook Member Function setCourseName**

The enhancement to class GradeBook is in the definition of setCourseName (Fig. 16.16, lines 16–29). The if statement in lines 18–19 determines whether parameter name contains a valid course name (i.e., a string of 25 or fewer characters). If the course name is valid, line 19 stores it in data member courseName. Note the expression name.length() in line 18. This is a member-function call just like myGradeBook.displayMessage(). The C++ Standard Library’s string class defines a member function length that returns the number of characters in a string object. Parameter name is a string object, so the call name.length() returns the number of characters in name. If this value is less than or equal to 25, name is valid and line 19 executes.

The if statement in lines 21–28 handles the case in which setCourseName receives an invalid course name (i.e., a name that is more than 25 characters long). Even if parameter
Validating Data with set Functions

name is too long, we still want to leave the GradeBook object in a consistent state—that is, a state in which the object’s data member courseName contains a valid value (i.e., a string of 25 characters or less). Thus, we truncate the specified course name and assign the first 25 characters of name to courseName (unfortunately, this could truncate the course name awkwardly). Standard class string provides member function substr (short for “substring”) that returns a new string object created by copying part of an
Chapter 16 Introduction to Classes and Objects

existing string object. The call in line 24 (i.e., `name.substr(0, 25)`) passes two integers (0 and 25) to `name`'s member function `substr`. These arguments indicate the portion of the string `name` that `substr` should return. The first argument specifies the starting position in the original string from which characters are copied—the first character in every string is considered to be at position 0. The second argument specifies the number of characters to copy. Therefore, the call in line 24 returns a 25-character substring of `name` starting at position 0 (i.e., the first 25 characters in `name`). For example, if `name` holds the value "CS101 Introduction to Programming in C++", `substr` returns "CS101 Introduction to Pro". After the call to `substr`, line 24 assigns the substring returned by `substr` to data member `courseName`. In this way, `setCourseName` ensures that `courseName` is always assigned a string containing 25 or fewer characters. If the member function has to truncate the course name to make it valid, lines 26–27 display a warning message.

The if statement in lines 21–28 contains two body statements—one to set the `courseName` to the first 25 characters of parameter `name` and one to print an accompanying message to the user. The statement in lines 26–27 could also appear without a stream insertion operator at the start of the second line of the statement, as in:

```
cout << "Name "" << name << "\" exceeds maximum length (25).\n"" Limiting courseName to first 25 characters.\n"" << endl;
```

The C++ compiler combines adjacent string literals, even if they appear on separate lines of a program. Thus, in the statement above, the C++ compiler would combine the string literals "Name "" << name << "\" exceeds maximum length (25).\n"" Limiting courseName to first 25 characters.\n"" into a single string literal that produces output identical to that of lines 26–27 in Fig. 16.16. This behavior allows you to print lengthy strings by breaking them across lines in your program without including additional stream insertion operations.

Testing Class GradeBook

Figure 16.17 demonstrates the modified version of class GradeBook (Figs. 16.15–16.16) featuring validation. Line 12 creates a GradeBook object named `gradeBook1`. Recall that the GradeBook constructor calls `setCourseName` to initialize data member `courseName`. In previous versions of the class, the benefit of calling `setCourseName` in the constructor was not evident. Now, however, the constructor takes advantage of the validation provided by `setCourseName`. The constructor simply calls `setCourseName`, rather than duplicating its validation code. When line 12 of Fig. 16.17 passes an initial course name of "CS101 Introduction to Programming in C++" to the GradeBook constructor, the constructor passes this value to `setCourseName`, where the actual initialization occurs. Because this course name contains more than 25 characters, the body of the second if statement executes, causing `courseName` to be initialized to the truncated 25-character course name "CS101 Introduction to Pro" (the truncated part is highlighted in darker blue in line 12). The output in Fig. 16.17 contains the warning message output by lines 26–27 of Fig. 16.16 in member function `setCourseName`. Line 13 creates another GradeBook object called `gradeBook2`—the valid course name passed to the constructor is exactly 25 characters.

Lines 16–19 of Fig. 16.17 display the truncated course name for `gradeBook1` (we highlight this in blue) and the course name for `gradeBook2`. Line 22 calls `gradeBook1`'s `setCourseName` member function directly, to change the course name in the GradeBook object
Validating Data with set Functions

to a shorter name that does not need to be truncated. Then, lines 25–28 output the course
names for the GradeBook objects again.

Additional Notes on Set Functions

A public set function such as setCourseName should carefully scrutinize any attempt to
modify the value of a data member (e.g., courseName) to ensure that the new value is ap-
propriate for that data item. For example, an attempt to set the day of the month to 37
should be rejected, an attempt to set a person’s weight to zero or a negative value should
be rejected, an attempt to set a grade on an exam to 185 (when the proper range is zero to
100) should be rejected, and so on.
Chapter 16 Introduction to Classes and Objects

Software Engineering Observation 16.6
Making data members private and controlling access, especially write access, to those data members through public member functions helps ensure data integrity.

Error-Prevention Tip 16.5
The benefits of data integrity are not automatic simply because data members are made private—you must provide appropriate validity checking and report the errors.

A class’s set functions can return values to the class’s clients indicating that attempts were made to assign invalid data to objects of the class. A client can test the return value of a set function to determine whether the attempt to modify the object was successful and to take appropriate action. In Chapter 24, we demonstrate how clients of a class can be notified via the exception-handling mechanism when an attempt is made to modify an object with an inappropriate value. To keep the program of Figs. 16.15–16.17 simple at this point in our C++ discussion, setCourseName in Fig. 16.16 just prints an appropriate message.

16.10 Wrap-Up

In this chapter, you created user-defined classes, and created and used objects of those classes. We declared data members of a class to maintain data for each object of the class. We also defined member functions that operate on that data. You learned how to call an object’s member functions to request the services the object provides and how to pass data to those member functions as arguments. We discussed the difference between a local variable of a member function and a data member of a class. We also showed how to use a constructor to specify initial values for an object’s data members. You learned how to separate the interface of a class from its implementation to promote good software engineering. We presented a diagram that shows the files that class-implementation programmers and client-code programmers need to compile the code they write. We demonstrated how set functions can be used to validate an object’s data and ensure that objects are maintained in a consistent state. UML class diagrams were used to model classes and their constructors, member functions and data members.

Summary

Section 16.2 Classes, Objects, Member Functions and Data Members
- Performing a task in a program requires a function. The function hides from its user the complex tasks that it performs.
- A function in a class is known as a member function and performs one of the class’s tasks.
- You must create an object of a class before a program can perform the tasks the class describes.
- Each message sent to an object is a member-function call that tells the object to perform a task.
- An object has attributes that are carried with the object as it’s used in a program. These attributes are specified as data members in the object’s class.

Section 16.3 Defining a Class with a Member Function
- A class definition contains the data members and member functions that define the class’s attributes and behaviors, respectively.
• A class definition begins with the keyword `class` followed immediately by the class name.
• By convention, the name of a user-defined class begins with a capital letter and, for readability, each subsequent word in the class name begins with a capital letter.
• Every class’s body is enclosed in a pair of braces (`{` and `}`) and ends with a semicolon.
• Member functions that appear after access specifier `public` can be called by other functions in a program and by member functions of other classes.
• Access specifiers are always followed by a colon (`:`).
• Keyword `void` is a special return type which indicates that a function will perform a task but will not return any data to its calling function when it completes its task.
• By convention, function names begin with a lowercase first letter and all subsequent words in the name begin with a capital letter.
• An empty set of parentheses after a function name indicates that the function does not require additional data to perform its task.
• Every function’s body is delimited by left and right braces (`{` and `}`).
• Typically, you cannot call a member function until you create an object of its class.
• Each new class you create becomes a new type in C++.
• In the UML, each class is modeled in a class diagram as a rectangle with three compartments. The top compartment contains the class name. The middle compartment contains the class’s attributes. The bottom compartment contains the class’s operations.
• The UML models operations as the operation name followed by parentheses. A plus sign (+) preceding the name indicates a `public` operation (i.e., a `public` member function in C++).

Section 16.4 Defining a Member Function with a Parameter
• A member function can require one or more parameters that represent additional data it needs to perform its task. A function call supplies arguments for each of the function’s parameters.
• A member function is called by following the object name with a dot operator (`.`), the function name and a set of parentheses containing the function’s arguments.
• A variable of C++ Standard Library class `string` represents a string of characters. This class is defined in header file `<string>`, and the name `string` belongs to namespace `std`.
• Function `getline` (from header `<string>`) reads characters from its first argument until a newline character is encountered, then places the characters (not including the newline) in the `string` variable specified as its second argument. The newline character is discarded.
• A parameter list may contain any number of parameters, including none at all (represented by empty parentheses) to indicate that a function does not require any parameters.
• The number of arguments in a function call must match the number of parameters in the parameter list of the called member function’s header. Also, the argument types in the function call must be consistent with the types of the corresponding parameters in the function header.
• The UML models a parameter of an operation by listing the parameter name, followed by a colon and the parameter type between the parentheses following the operation name.
• The UML has its own data types. Not all the UML data types have the same names as the corresponding C++ types. The UML type `String` corresponds to the C++ type `string`.

Section 16.5 Data Members, `set` Functions and `get` Functions
• Variables declared in a function’s body are local variables and can be used only from the point of their declaration in the function to the immediately following closing right brace (`}`). When a function terminates, the values of its local variables are lost.
• A local variable must be declared before it can be used in a function. A local variable cannot be accessed outside the function in which it’s declared.

• Data members normally are private. Variables or functions declared private are accessible only to member functions of the class in which they’re declared, or to friends of the class.

• When a program creates (instantiates) an object of a class, its private data members are encapsulated (hidden) in the object and can be accessed only by member functions of the object’s class.

• When a function that specifies a return type other than void is called and completes its task, the function returns a result to its calling function.

• By default, the initial value of a string is the empty string—i.e., a string that does not contain any characters. Nothing appears on the screen when an empty string is displayed.

• Classes often provide public member functions to allow clients of the class to set or get private data members. The names of these member functions normally begin with set or get.

• Set and get functions allow clients of a class to indirectly access the hidden data. The client does not know how the object performs these operations.

• A class’s set and get functions should be used by other member functions of the class to manipulate the class’s private data. If the class’s data representation is changed, member functions that access the data only via the set and get functions will not require modification.

• A public set function should carefully scrutinize any attempt to modify the value of a data member to ensure that the new value is appropriate for that data item.

• The UML represents data members as attributes by listing the attribute name, followed by a colon and the attribute type. Private attributes are preceded by a minus sign (–) in the UML.

• The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name.

• UML class diagrams do not specify return types for operations that do not return values.

Section 16.6 Initializing Objects with Constructors

• Each class should provide a constructor to initialize an object of the class when the object is created. A constructor must be defined with the same name as the class.

• A difference between constructors and functions is that constructors cannot return values, so they cannot specify a return type (not even void). Normally, constructors are declared public.

• C++ requires a constructor call at the time each object is created, which helps ensure that every object is initialized before it’s used in a program.

• A constructor with no parameters is a default constructor. If you do not provide a constructor, the compiler provides a default constructor. You can also define a default constructor explicitly. If you define a constructor for a class, C++ will not create a default constructor.

• The UML models constructors as operations in a class diagram’s third compartment with the word “constructor” between guillemets (« and ») before the constructor’s name.

Section 16.7 Placing a Class in a Separate File for Reusability

• Class definitions, when packaged properly, can be reused by programmers worldwide.

• It’s customary to define a class in a header file that has a .h filename extension.

• If the class’s implementation changes, the class’s clients should not be required to change.

• Interfaces define and standardize the ways in which things such as people and systems interact.

• A class’s public interface describes the public member functions that are made available to the class’s clients. The interface describes what services clients can use and how to request those services, but does not specify how the class carries out the services.
Section 16.8 Separating Interface from Implementation

- Separating interface from implementation makes programs easier to modify. Changes in the class’s implementation do not affect the client as long as the class’s interface remains unchanged.
- A function prototype contains a function’s name, its return type and the number, types and order of the parameters the function expects to receive.
- Once a class is defined and its member functions are declared (via function prototypes), the member functions should be defined in a separate source-code file.
- For each member function defined outside of its corresponding class definition, the function name must be preceded by the class name and the binary scope resolution operator (::).

Section 16.9 Validating Data with set Functions

- Class string’s length member function returns the number of characters in a string object.
- Class string’s member function substr returns a new string object containing a copy of part of an existing string object. The first argument specifies the starting position in the original string. The second argument specifies the number of characters to copy.

Terminology

- access specifier: 564
- access-specifier label public: 564
- accessor: 574
- argument: 566
- binary scope resolution operator (::): 586
- body of a class definition: 563
- calling function: 564
- camel case: 563
- class definition: 563
- class keyword: 563
- client of an object: 574
- consistent state: 591
- constructor: 576
- data hiding: 572
- data member: 569
- default constructor: 576
- defining a class: 563
- dot operator (.): 565
- driver program: 580
- empty string: 574
- extensible language: 565
- function header: 564
- function prototype: 584
- get function: 574
- getline function of <string> library: 567
- header file: 580
- interface of a class: 584
- interface: 583
- length member function of class string: 590
- local variable: 569
- member-function call: 562
- message (send to an object): 562
- mutator: 574
- parameter: 566
- parameter list: 568
- private: access specifier: 571
- public access specifier: 564
- public services of a class: 584
- request a service from an object: 562
- return statement: 573
- return type: 564
- separate interface from implementation: 583
- set function: 574
- source-code file: 580
- string class: 567
- <string> header file: 567
- substr member function of class string: 591
- UML class diagram: 565
- user-defined type: 565
- validation: 590
- validity checking: 590
- void return type: 564

Self-Review Exercises

16.1 Fill in the blanks in each of the following:

a) A house is to a blueprint as a(n) ______ is to a class.

b) Every class definition contains the keyword ______ followed immediately by the class’s name.
c) A class definition is typically stored in a file with the ______ filename extension.
d) Each parameter in a function header must specify both a(n) ______ and a(n) ______.
e) When each object of a class maintains its own copy of an attribute, the variable that represents the attribute is also known as a(n) ______.
f) Keyword ______ is a(n) ______.
g) Return type ______ indicates that a function will perform a task but will not return any information when it completes its task.
h) Function ______ from the <string> library reads characters until a newline character is encountered, then copies those characters into the specified string.
i) When a member function is defined outside the class definition, the function header must include the class name and the ______, followed by the function name to “tie” the member function to the class definition.
j) The source-code file and any other files that use a class can include the class’s header file via a(n) ______ preprocessor directive.

16.2 State whether each of the following is true or false. If false, explain why.
a) By convention, function names begin with a capital letter and all subsequent words in the name begin with a capital letter.
b) Empty parentheses following a function name in a function prototype indicate that the function does not require any parameters to perform its task.
c) Data members or member functions declared with access specifier ______ are accessible to member functions of the class in which they’re declared.
d) Variables declared in the body of a particular member function are known as data members and can be used in all member functions of the class.
e) Every function’s body is delimited by left and right braces ({}).
f) Any source-code file that contains int main() can be used to execute a program.
g) The types of arguments in a function call must be consistent with the types of the corresponding parameters in the function prototype’s parameter list.

16.3 What is the difference between a local variable and a data member?

16.4 Explain the purpose of a function parameter. What’s the difference between a parameter and an argument?

Answers to Self-Review Exercises

16.1 a) object. b) class. c) .h. d) type, name. e) data member. f) access specifier. g) void. h) getline. i) binary scope resolution operator (::). j) #include.

16.2 a) False. Function names begin with a lowercase letter and all subsequent words in the name begin with a capital letter. b) True. c) True. d) False. Such variables are local variables and can be used only in the member function in which they’re declared. e) True. f) True. g) True.

16.3 A local variable is declared in the body of a function and can be used only from the point at which it’s declared to the closing brace of the block in which it’s declared. A data member is declared in a class, but not in the body of any of the class’s member functions. Every object of a class has a separate copy of the class’s data members. Data members are accessible to all member functions of the class.

16.4 A parameter represents additional information that a function requires to perform its task. Each parameter required by a function is specified in the function header. An argument is the value supplied in the function call. When the function is called, the argument value is passed into the function parameter so that the function can perform its task.
Exercises

16.5  Explain the difference between a function prototype and a function definition.

16.6  What's a default constructor? How are an object’s data members initialized if a class has only an implicitly defined default constructor?

16.7  Explain the purpose of a data member.

16.8  What’s a header file? What’s a source-code file? Discuss the purpose of each.

16.9  Explain how a program could use class string without inserting a using declaration.

16.10 Explain why a class might provide a set function and a get function for a data member.

16.11 (Modifying Class GradeBook) Modify class GradeBook (Figs. 16.11–16.12) as follows:

a) Include a second string data member that represents the course instructor’s name.

b) Provide a set function to change the instructor’s name and a get function to retrieve it.

c) Modify the constructor to specify course name and instructor name parameters.

d) Modify function displayMessage to output the welcome message and course name, then the string "This course is presented by: " followed by the instructor's name.

16.12 (Account Class) Create an Account class that a bank might use to represent customers’ bank accounts. Include a data member of type int to represent the account balance. Provide a constructor that receives an initial balance and uses it to initialize the data member. The constructor should validate the initial balance to ensure that it’s greater than or equal to 0. If not, set the balance to 0 and display an error message indicating that the initial balance was invalid. Provide three member functions. Member function credit should add an amount to the current balance. Member function debit should withdraw money from the Account and ensure that the debit amount does not exceed the Account’s balance. If it does, the balance should be left unchanged and the function should print a message indicating "Debit amount exceeded account balance." Member function getBalance should return the current balance. Create a program that creates two Account objects and tests the member functions of class Account.

16.13 (Invoice Class) Create a class called Invoice that a hardware store might use to represent an invoice for an item sold at the store. An Invoice should include four data members—a part number (type string), a part description (type string), a quantity of the item being purchased (type int) and a price per item (type int). Your class should have a constructor that initializes the four data members. Provide a set and a get function for each data member. In addition, provide a member function named getInvoiceAmount that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as an int value. If the quantity is not positive, it should be set to 0. If the price per item is not positive, it should be set to 0. Write a test program that demonstrates class Invoice’s capabilities.

16.14 (Employee Class) Create a class called Employee that includes three pieces of information as data members—a first name (type string), a last name (type string) and a monthly salary (type int). Your class should have a constructor that initializes the three data members. Provide a set and a get function for each data member. If the monthly salary is not positive, set it to 0. Write a test program that demonstrates class Employee’s capabilities. Create two Employee objects and display each object’s yearly salary. Then give each Employee a 10 percent raise and display each Employee’s yearly salary again.

16.15 (Date Class) Create a class called Date that includes three pieces of information as data members—a month (type int), a day (type int) and a year (type int). Your class should have a constructor with three parameters that uses the parameters to initialize the three data members. For the
purpose of this exercise, assume that the values provided for the year and day are correct, but ensure that the month value is in the range 1–12; if it isn’t, set the month to 1. Provide a set and a get function for each data member. Provide a member function displayDate that displays the month, day and year separated by forward slashes (/). Write a test program that demonstrates class Date’s capabilities.

Making a Difference

16.16 (Target-Heart-Rate Calculator) While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your trainers and doctors. According to the American Heart Association (AHA) (www.americanheart.org/presenter.jhtml?identifier=4736), the formula for calculating your maximum heart rate in beats per minute is 220 minus your age in years. Your target heart rate is a range that is 50–85% of your maximum heart rate. [Note: These formulas are estimates provided by the AHA. Maximum and target heart rates may vary based on the health, fitness and gender of the individual. Always consult a physician or qualified health care professional before beginning or modifying an exercise program.] Create a class called HeartRates. The class attributes should include the person’s first name, last name and date of birth (consisting of separate attributes for the month, day and year of birth). Your class should have a constructor that receives this data as parameters. For each attribute provide set and get functions. The class also should include a function getAge that calculates and returns the person’s age (in years), a function getMaxiumumHeartRate that calculates and returns the person’s maximum heart rate and a function getTargetHeartRate that calculates and returns the person’s target heart rate. Since you do not yet know how to obtain the current date from the computer, function getAge should prompt the user to enter the current month, day and year before calculating the person’s age. Write an application that prompts for the person’s information, instantiates an object of class HeartRates and prints the information from that object—including the person’s first name, last name and date of birth—then calculates and prints the person’s age in (years), maximum heart rate and target-heart-rate range.

16.17 (Computerization of Health Records) A health care issue that has been in the news lately is the computerization of health records. This possibility is being approached cautiously because of sensitive privacy and security concerns, among others. [We address such concerns in later exercises.] Computerizing health records could make it easier for patients to share their health profiles and histories among their various health care professionals. This could improve the quality of health care, help avoid drug conflicts and erroneous drug prescriptions, reduce costs and in emergencies, could save lives. In this exercise, you’ll design a “starter” HealthProfile class for a person. The class attributes should include the person’s first name, last name, gender, date of birth (consisting of separate attributes for the month, day and year of birth), height (in inches) and weight (in pounds). Your class should have a constructor that receives this data. For each attribute, provide set and get functions. The class also should include functions that calculate and return the user’s age in years, maximum heart rate and target-heart-rate range (see Exercise 16.16), and body mass index (BMI; see Exercise 2.32). Write an application that prompts for the person’s information, instantiates an object of class HealthProfile for that person and prints the information from that object—including the person’s first name, last name, gender, date of birth, height and weight—then calculates and prints the person’s age in years, BMI, maximum heart rate and target-heart-rate range. It should also display the “BMI values” chart from Exercise 2.32. Use the same technique as Exercise 16.16 to calculate the person’s age.
My object all sublime
I shall achieve in time.
—W. S. Gilbert

Is it a world to hide virtues in?
—William Shakespeare

Don’t be “consistent,” but be simply true.
—Oliver Wendell Holmes, Jr.

Objectives
In this chapter you’ll learn:

■ How to use a preprocessor wrapper to prevent multiple definition errors.
■ To understand class scope and accessing class members via the name of an object, a reference to an object or a pointer to an object.
■ To define constructors with default arguments.
■ How destructors are used to perform “termination housekeeping” on an object before it’s destroyed.
■ When constructors and destructors are called and the order in which they’re called.
■ The logic errors that may occur when a public member function returns a reference to private data.
■ To assign the data members of one object to those of another object by default memberwise assignment.
17.1 Introduction

In Chapters 15–16, we introduced many basic terms and concepts of C++ object-oriented programming. We also discussed our program development methodology: We selected appropriate attributes and behaviors for each class and specified the manner in which objects of our classes collaborated with objects of C++ Standard Library classes to accomplish each program’s overall goals.

In this chapter, we take a deeper look at classes. We use an integrated Time class case study in both this chapter and Chapter 18, Classes: A Deeper Look, Part 2 to demonstrate several class construction capabilities. We begin with a Time class that reviews several of the features presented in the preceding chapters. The example also demonstrates an important C++ software engineering concept—using a “preprocessor wrapper” in header files to prevent the code in the header from being included into the same source code file more than once. Since a class can be defined only once, using such preprocessor directives prevents multiple definition errors.

Next, we discuss class scope and the relationships among class members. We demonstrate how client code can access a class’s public members via three types of “handles” — the name of an object, a reference to an object or a pointer to an object. As you’ll see, object names and references can be used with the dot (.) member selection operator to access a public member, and pointers can be used with the arrow (->) member selection operator.

We discuss access functions that can read or display data in an object. A common use of access functions is to test the truth or falsity of conditions — such functions are known as predicate functions. We also demonstrate the notion of a utility function (also called a helper function) — a private member function that supports the operation of the class’s public member functions, but is not intended for use by clients of the class.

In the second Time class case study example, we demonstrate how to pass arguments to constructors and show how default arguments can be used in a constructor to enable client code to initialize objects using a variety of arguments. Next, we discuss a special member function called a destructor that is part of every class and is used to perform “termination housekeeping” on an object before the object is destroyed. We then demonstrate the order in which constructors and destructors are called, because your programs’ correctness depends on using properly initialized objects that have not yet been destroyed.
Our last example of the Time class case study in this chapter shows a dangerous programming practice in which a member function returns a reference to private data. We discuss how this breaks the encapsulation of a class and allows client code to directly access an object’s data. This last example shows that objects of the same class can be assigned to one another using default memberwise assignment, which copies the data members in the object on the right side of the assignment into the corresponding data members of the object on the left side of the assignment. The chapter concludes with a discussion of software reusability.

17.2 Time Class Case Study

Our first example (Figs. 17.1–17.3) creates class Time and a driver program that tests the class. In this section, we review many of the concepts covered in Chapter 16 and demonstrate an important C++ software engineering concept—using a “preprocessor wrapper” in header files to prevent the code in the header from being included into the same source code file more than once. Since a class can be defined only once, using such preprocessor directives prevents multiple-definition errors.

```cpp
// Fig. 17.1: Time.h
// Declaration of class Time.
// Member functions are defined in Time.cpp

// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H

// Time class definition
class Time
{
  public:
    Time(); // constructor
    void setTime( int, int, int ); // set hour, minute and second
    void printUniversal(); // print time in universal-time format
    void printStandard(); // print time in standard-time format
  private:
    int hour; // 0 - 23 (24-hour clock format)
    int minute; // 0 - 59
    int second; // 0 - 59
}; // end class Time

#endif
```

Fig. 17.1 | Time class definition.

**Time Class Definition**
The class definition (Fig. 17.1) contains prototypes (lines 13–16) for member functions Time, setTime, printUniversal and printStandard, and includes private integer members hour, minute and second (lines 18–20). Class Time’s private data members can be accessed only by its four member functions. Chapter 20 introduces a third access specifier, protected, as we study inheritance and the part it plays in object-oriented programming.
In Fig. 17.1, the class definition is enclosed in the following preprocessor wrapper (lines 6, 7 and 23):

```c
// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H
...
#endif
```

When we build larger programs, other definitions and declarations will also be placed in header files. The preceding preprocessor wrapper prevents the code between `#ifndef` (which means “if not defined”) and `#endif` from being included if the name `TIME_H` has been defined. If the header has not been included previously in a file, the name `TIME_H` is defined by the `#define` directive and the header file statements are included. If the header has been included previously, `TIME_H` is defined already and the header file is not included again. Attempts to include a header file multiple times (inadvertently) typically occur in large programs with many header files that may themselves include other header files. [Note: The commonly used convention for the symbolic constant name in the preprocessor directives is simply the header file name in uppercase with the underscore character replacing the period.]

**Time Class Member Functions**

In Fig. 17.2, the `Time` constructor (lines 10–13) initializes the data members to 0—the universal-time equivalent of 12 AM. This ensures that the object begins in a consistent state. Invalid values cannot be stored in the data members of a `Time` object, because the constructor is called when the `Time` object is created, and all subsequent attempts by a client to modify the data members are scrutinized by function `setTime` (discussed shortly). You can define several overloaded constructors for a class.

The data members of a class cannot be initialized where they’re declared in the class body. It’s strongly recommended that these data members be initialized by the class’s constructor (as there is no default initialization for fundamental-type data members). Data members can also be assigned values by `Time`’s `set` functions. [Note: Chapter 18 demon-
Fig. 17.2: Time.cpp
// Member-function definitions for class Time.
#include <iostream>
#include <iomanip>
#include "Time.h" // include definition of class Time from Time.h
using namespace std;

Time::Time()
{
    hour = minute = second = 0;
} // end Time constructor

void Time::setTime( int h, int m, int s )
{
    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setTime

void Time::printUniversal()
{
    cout << setfill( '0' ) << setw( 2 ) << hour << ":" << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

void Time::printStandard()
{
    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << "":";
    << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard

strates that only a class’s static const data members of integral or enum types can be initialized in the class’s body.]

Common Programming Error 17.1
Attempting to initialize a non-static data member of a class explicitly in the class definition is a syntax error.

Function setTime (lines 17–22) is a public function that declares three int parameters and uses them to set the time. A conditional expression tests each argument to determine whether the value is in a specified range. For example, the hour value (line 19) must be greater than or equal to 0 and less than 24, because the universal-time format represents hours as integers from 0 to 23 (e.g., 1 PM is hour 13 and 11 PM is hour 23; midnight is hour 0 and noon is hour 12). Similarly, both minute and second values (lines 20 and 21)
must be greater than or equal to 0 and less than 60. Any values outside these ranges are set
to zero to ensure that a Time object always contains consistent data—that is, the object’s data
values are always kept in range, even if the values provided as arguments to function setTime
were incorrect. In this example, zero is a consistent value for hour, minute and second.

A value passed to setTime is a correct value if it’s in the allowed range for the member
it’s initializing. So, any number in the range 0–23 would be a correct value for the hour.
A correct value is always a consistent value. However, a consistent value is not necessarily
a correct value. If setTime sets hour to 0 because the argument received was out of range,
then hour is correct only if the current time is coincidentally midnight.

Function printUniversal (lines 25–29 of Fig. 17.2) takes no arguments and outputs
the time in universal-time format, consisting of three colon-separated pairs of digits for the
hour, minute and second. For example, if the time were 1:30:07 PM, function printUni-
versal would return 13:30:07. Line 27 uses parameterized stream manipulator setfill
to specify the fill character that is displayed when an integer is output in a field wider than
the number of digits in the value. By default, the fill characters appear to the left of the
digits in the number. In this example, if the minute value is 2, it will be displayed as 02,
because the fill character is set to zero (‘0’). If the number being output fills the specified
field, the fill character will not be displayed. Once the fill character is specified with set-
fill, it applies for all subsequent values that are displayed in fields wider than the value
being displayed (i.e., setfill is a “sticky” setting). This is in contrast to setw, which
applies only to the next value displayed (setw is a “nonsticky” setting).

Error-Prevention Tip 17.2
Each sticky setting (such as a fill character or floating-point precision) should be restored
to its previous setting when it’s no longer needed. Failure to do so may result in incorrectly
formatted output later in a program. Chapter 23, Stream Input/Output, discusses how to
reset the fill character and precision.

Function printStandard (lines 32–37) takes no arguments and outputs the date in
standard-time format, consisting of the hour, minute and second values separated by
colons and followed by an AM or PM indicator (e.g., 1:27:06 PM). Like function print-
Universal, function printStandard uses setfill(‘0’) to format the minute and second
as two digit values with leading zeros if necessary. Line 34 uses the conditional operator
(?:) to determine the value of hour to be displayed—if the hour is 0 or 12 (AM or PM),
it appears as 12; otherwise, the hour appears as a value from 1 to 11. The conditional oper-
ator in line 36 determines whether AM or PM will be displayed.

Defining Member Functions Outside the Class Definition; Class Scope
Even though a member function declared in a class definition may be defined outside that
class definition (and “tied” to the class via the binary scope resolution operator), that mem-
er function is still within that class’s scope; i.e., its name is known only to other members
of the class unless referred to via an object of the class, a reference to an object of the class,
a pointer to an object of the class or the binary scope resolution operator. We’ll say more
about class scope shortly.

If a member function is defined in the body of a class definition, the compiler
attempts to inline calls to the member function. Remember that the compiler reserves the
right not to inline any function.
Member Functions vs. Global Functions
The printUniversal and printStandard member functions take no arguments, because these member functions implicitly know that they’re to print the data members of the particular Time object for which they’re invoked. This can make member function calls more concise than conventional function calls in procedural programming.

Using Class Time
Once class Time has been defined, it can be used as a type in object, array, pointer and reference declarations as follows:

```cpp
Time sunset; // object of type Time
Time arrayOfTimes[5]; // array of 5 Time objects
Time &dinnerTime = sunset; // reference to a Time object
Time *timePtr = &dinnerTime; // pointer to a Time object
```
Figure 17.3 uses class Time. Line 10 instantiates a single object of class Time called t. When the object is instantiated, the Time constructor is called to initialize each private data member to 0. Then, lines 14 and 16 print the time in universal and standard formats, respectively, to confirm that the members were initialized properly. Line 18 sets a new time by calling member function setTime, and lines 22 and 24 print the time again in universal and standard formats, respectively, to confirm that the new value was set.

```
// Fig. 17.3: fig17_03.cpp
// Program to test class Time.
// NOTE: This file must be compiled with Time.cpp.
#include <iostream>
#include "Time.h" // include definition of class Time from Time.h
using namespace std;

int main()
{
    Time t; // instantiate object t of class Time

    // output Time object t's initial values
    cout << "The initial universal time is ";
t.printUniversal(); // 00:00:00
    cout << "\nThe initial standard time is ";
t.printStandard(); // 12:00:00 AM

    t.setTime( 13, 27, 6 ); // change time

    // output Time object t's new values
    cout << "\nUniversal time after setTime is ";
t.printUniversal(); // 13:27:06
    cout << "\nStandard time after setTime is ";
t.printStandard(); // 1:27:06 PM

    t.setTime( 99, 99, 99 ); // attempt invalid settings

    // output t's values after specifying invalid values
    cout << "\nAfter attempting invalid settings:
    " << "\nUniversal time: ";
t.printUniversal(); // 00:00:00
    cout << "\nStandard time: ";
t.printStandard(); // 12:00:00 AM
    cout << endl;
} // end main
```

The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
both formats. Line 26 attempts to use `setTime` to set the data members to invalid values—function `setTime` recognizes this and sets the invalid values to 0 to maintain the object in a consistent state. Finally, lines 31 and 33 print the time again in both formats.

**Looking Ahead to Composition and Inheritance**

Often, classes do not have to be created “from scratch.” Rather, they can include objects of other classes as members or they may be derived from other classes that provide attributes and behaviors the new classes can use. Such software reuse can greatly enhance productivity and simplify code maintenance. Including class objects as members of other classes is called composition (or aggregation) and is discussed in Chapter 18. Deriving new classes from existing classes is called inheritance and is discussed in Chapter 20.

**Object Size**

People new to object-oriented programming often suppose that objects must be quite large because they contain data members and member functions. Logically, this is true—you may think of objects as containing data and functions (and our discussion has certainly encouraged this view); physically, however, this is not true.

Performance Tip 17.2

Objects contain only data, so objects are much smaller than if they also contained member functions. Applying operator `sizeof` to a class name or to an object of that class will report only the size of the class’s data members. The compiler creates one copy (only) of the member functions separate from all objects of the class. All objects of the class share this one copy. Each object, of course, needs its own copy of the class’s data, because the data can vary among the objects. The function code is nonmodifiable and, hence, can be shared among all objects of one class.

**17.3 Class Scope and Accessing Class Members**

A class’s data members (variables declared in the class definition) and member functions (functions declared in the class definition) belong to that class’s scope. Nonmember functions are defined at global namespace scope.

Within a class’s scope, class members are immediately accessible by all of that class’s member functions and can be referenced by name. Outside a class’s scope, public class members are referenced through one of the handles on an object—an object name, a reference to an object or a pointer to an object. The type of the object, reference or pointer specifies the interface (i.e., the member functions) accessible to the client. [We’ll see in Chapter 18 that an implicit handle is inserted by the compiler on every reference to a data member or member function from within an object.]

Member functions of a class can be overloaded, but only by other member functions of that class. To overload a member function, simply provide in the class definition a prototype for each version of the overloaded function, and provide a separate function definition for each version of the function.

Variables declared in a member function have local scope and are known only to that function. If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the block-scope variable in the local scope. Such a hidden variable can be accessed by preceding the variable name with the class name
Chapter 17  Classes: A Deeper Look, Part 1

followed by the scope resolution operator (::). Hidden global variables can be accessed with the unary scope resolution operator (see Chapter 15).

The dot member selection operator (.) is preceded by an object’s name or with a reference to an object to access the object’s members. The arrow member selection operator (->) is preceded by a pointer to an object to access the object’s members.

Figure 17.4 uses a simple class called Count (lines 7–24) with private data member x of type int (line 23), public member function setX (lines 11–14) and public member function print (lines 17–20) to illustrate accessing class members with the member-selection operators. For simplicity, we’ve included this small class in the same file as main. Lines 28–30 create three variables related to type Count—counter (a Count object), counterPtr (a pointer to a Count object) and counterRef (a reference to a Count object). Variable counterRef refers to counter, and variable counterPtr points to counter. In lines 33–34 and 37–38, the program can invoke member functions setX and print by using the dot (.) member selection operator preceded by either the name of the object (counter) or a reference to the object (counterRef, which is an alias for counter). Similarly, lines 41–42 demonstrate that the program can invoke member functions setX and print by using a pointer (counterPtr) and the arrow (->) member-selection operator.

```cpp
1 // Fig. 17.4: fig17_04.cpp
2 // Demonstrating the class member access operators . and ->
3 #include <iostream>
4 using namespace std;
5
6 // class Count definition
7 class Count
8 {
9     public:  // public data is dangerous
10         // sets the value of private data member x
11         void setX( int value )
12         {
13             x = value;
14         } // end function setX
15
16         // prints the value of private data member x
17         void print()
18         {
19             cout << x << endl;
20         } // end function print
21
22     private:
23         int x;
24     }; // end class Count
25
26 int main()
27 {
28     Count counter;  // create counter object
29     Count *counterPtr = &counter;  // create pointer to counter
30     Count &counterRef = counter;  // create reference to counter
```

Fig. 17.4  Accessing an object’s member functions through each type of object handle—the object’s name, a reference to the object and a pointer to the object. (Part 1 of 2.)
17.4 Separating Interface from Implementation

In Chapter 16, we began by including a class's definition and member-function definitions in one file. We then demonstrated separating this code into two files—a header file for the class definition (i.e., the class's interface) and a source code file for the class's member-function definitions (i.e., the class's implementation). Recall that this makes it easier to modify programs—as far as clients of a class are concerned, changes in the class's implementation do not affect the client as long as the class's interface originally provided to the client remains unchanged.

Actually, things are not quite this rosy. Header files do contain some portions of the implementation and hints about others. Inline member functions, for example, should be in a header file, so that when the compiler compiles a client, the client can include the inline function definition in place. A class's private members are listed in the class definition in the header file, so these members are visible to clients even though the clients may not access the private members. In Chapter 18, we show how to use a “proxy class” to hide even the private data of a class from clients of the class.

Software Engineering Observation 17.6

Clients of a class do not need access to the class's source code to use the class. The clients do, however, need to be able to link to the class's object code (i.e., the compiled version of the class). This encourages independent software vendors (ISVs) to provide class libraries for sale or license. The ISVs provide in their products only the header files and the object modules. No proprietary information is revealed—as would be the case if source code were provided. The C++ user community benefits by having more ISV-produced class libraries available.

Fig. 17.4 | Accessing an object's member functions through each type of object handle—the object's name, a reference to the object and a pointer to the object. (Part 2 of 2.)

```cpp
31 cout << "Set x to 1 and print using the object's name: ";
32 counter.setX( 1 ); // set data member x to 1
33 counter.print(); // call member function print
34
35 cout << "Set x to 2 and print using a reference to an object: ";
36 counterRef.setX( 2 ); // set data member x to 2
37 counterRef.print(); // call member function print
38
39 cout << "Set x to 3 and print using a pointer to an object: ";
40 counterPtr->setX( 3 ); // set data member x to 3
41 counterPtr->print(); // call member function print
42 } // end main
```
17.5 Access Functions and Utility Functions

Access functions can read or display data. Another common use for access functions is to test the truth or falsity of conditions—such functions are often called predicate functions. An example of a predicate function would be an isEmpty function for any container class—a class capable of holding many objects, like a vector. A program might test isEmpty before attempting to read another item from the container object. An isFull predicate function might test a container-class object to determine whether it has no additional room. Useful predicate functions for our Time class might be isAM and isPM.

The program of Figs. 17.5–17.7 demonstrates the notion of a utility function (also called a helper function). A utility function is not part of a class’s public interface; rather, it’s a private member function that supports the operation of the class’s public member functions. Utility functions are not intended to be used by clients of a class (but can be used by friends of a class, as we’ll see in Chapter 18).

Class SalesPerson (Fig. 17.5) declares an array of 12 monthly sales figures (line 17) and the prototypes for the class’s constructor and member functions that manipulate the array.

```cpp
// Fig. 17.5: SalesPerson.h
// SalesPerson class definition.
// Member functions defined in SalesPerson.cpp.
#ifndef SALESP_H
#define SALESP_H

class SalesPerson
{
public:
    static const int monthsPerYear = 12; // months in one year
    SalesPerson(); // constructor
    void getSalesFromUser(); // input sales from keyboard
    void setSales(int, double); // set sales for a specific month
    void printAnnualSales(); // summarize and print sales

private:
    double totalAnnualSales(); // prototype for utility function
    double sales[monthsPerYear]; // 12 monthly sales figures
}; // end class SalesPerson

#endif
```

Fig. 17.5 | SalesPerson class definition.

In Fig. 17.6, the SalesPerson constructor (lines 9–13) initializes array sales to zero. The public member function setSales (lines 30–37) sets the sales figure for one month...
in array sales. The public member function printAnnualSales (lines 40–45) prints the
total sales for the last 12 months. The private utility function totalAnnualSales (lines
48–56) totals the 12 monthly sales figures for the benefit of printAnnualSales. Member
function printAnnualSales edits the sales figures into monetary format.

```cpp
// Fig. 17.6: SalesPerson.cpp
// SalesPerson class member-function definitions.
#include <iostream>
#include <iomanip>
#include "SalesPerson.h" // include SalesPerson class definition
using namespace std;

// initialize elements of array sales to 0.0
SalesPerson::SalesPerson()
{
    for ( int i = 0; i < monthsPerYear; i++ )
        sales[ i ] = 0.0;
} // end SalesPerson constructor

// get 12 sales figures from the user at the keyboard
void SalesPerson::getSalesFromUser()
{
    double salesFigure;
    for ( int i = 1; i <= monthsPerYear; i++ )
    {
        cout << "Enter sales amount for month " << i << ": ";
        cin >> salesFigure;
        setSales( i, salesFigure );
    } // end for
} // end function getSalesFromUser

// set one of the 12 monthly sales figures; function subtracts
// one from month value for proper subscript in sales array
void SalesPerson::setSales( int month, double amount )
{
    // test for valid month and amount values
    if ( month >= 1 && month <= monthsPerYear && amount > 0 )
        sales[ month - 1 ] = amount; // adjust for subscripts 0-11
    else // invalid month or amount value
        cout << "Invalid month or sales figure" << endl;
} // end function setSales

// print total annual sales (with the help of utility function)
void SalesPerson::printAnnualSales()
{
    cout << setprecision( 2 ) << fixed
         << "The total annual sales are: $" << totalAnnualSales() << endl; // call utility function
} // end function printAnnualSales
```

Fig. 17.6 | SalesPerson class member-function definitions. (Part 1 of 2.)
Chapter 17  Classes: A Deeper Look, Part I

In Fig. 17.7, notice that the application’s main function includes only a simple sequence of member-function calls—there are no control statements. The logic of manipulating the sales array is completely encapsulated in class SalesPerson’s member functions.

Software Engineering Observation 17.8

A phenomenon of object-oriented programming is that once a class is defined, creating and manipulating objects of that class often involve issuing only a simple sequence of member-function calls—few, if any, control statements are needed. By contrast, it’s common to have control statements in the implementation of a class’s member functions.
The program of Figs. 17.8–17.10 enhances class $\text{Time}$ to demonstrate how arguments are implicitly passed to a constructor. The constructor defined in Fig. 17.2 initialized $\text{hour}$, $\text{minute}$ and $\text{second}$ to 0 (i.e., midnight in universal time). Like other functions, constructors can specify default arguments. Line 13 of Fig. 17.8 declares the $\text{Time}$ constructor to include default arguments, specifying a default value of zero for each argument passed to the constructor. In Fig. 17.9, lines 10–13 define the new version of the $\text{Time}$ constructor that receives values for parameters $\text{hr}$, $\text{min}$ and $\text{sec}$ that will be used to initialize private data members $\text{hour}$, $\text{minute}$ and $\text{second}$, respectively. Class $\text{Time}$ provides $\text{set}$ and $\text{get}$ functions for each data member. The $\text{Time}$ constructor now calls $\text{setTime}$, which calls the $\text{setHour}$, $\text{setMinute}$ and $\text{setSecond}$ functions to validate and assign values to the data members. The default arguments to the constructor ensure that, even if no values are provided in a constructor call, the constructor still initializes the data members to maintain the $\text{Time}$ object in a consistent state. A constructor that defaults all its arguments is also a default constructor—i.e., a constructor that can be invoked with no arguments. There can be at most one default constructor per class.

```cpp
// Fig. 17.8: Time.h
// Time class containing a constructor with default arguments.
// Member functions defined in Time.cpp.

// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H

// Time abstract data type definition
class Time {
public:
    Time( int hr = 0, int min = 0, int sec = 0 ); // default constructor

    // set functions
    void setTime( int, int, int ); // set hour, minute, second
    void setHour( int ); // set hour (after validation)
    void setMinute( int ); // set minute (after validation)
    void setSecond( int ); // set second (after validation)

    // get functions
    int getHour(); // return hour

#endif // TIME_H
```

**Fig. 17.7** Utility function demonstration. (Part 2 of 2.)

**17.6 Time Class Case Study: Constructors with Default Arguments**

```
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92
The total annual sales are: $60120.59
```
In Fig. 17.9, line 12 of the constructor calls member function `setTime` with the values passed to the constructor (or the default values). Function `setTime` calls `setHour` to ensure that the value supplied for `hour` is in the range 0–23, then calls `setMinute` and `setSecond` to ensure that the values for `minute` and `second` are each in the range 0–59. If a value is out of range, that value is set to zero (to ensure that each data member remains in a consistent state). In Chapter 24, Exception Handling, we use exceptions to indicate when a value is out of range, rather than simply assigning a default consistent value.

```cpp
23 int getMinute(); // return minute
24 int getSecond(); // return second
26 void printUniversal(); // output time in universal-time format
27 void printStandard(); // output time in standard-time format
private:
29 int hour; // 0 - 23 (24-hour clock format)
30 int minute; // 0 - 59
31 int second; // 0 - 59
}; // end class Time

```
The `Time` constructor could be written to include the same statements as member function `setTime`, or even the individual statements in the `setHour`, `setMinute` and `setSecond` functions. Calling `setHour`, `setMinute` and `setSecond` from the constructor may
be slightly more efficient because the extra call to `setTime` would be eliminated. Similarly, copying the code from lines 27, 33 and 39 into constructor would eliminate the overhead of calling `setTime`, `setHour`, `setMinute` and `setSecond`. Coding the `Time` constructor or member function `setTime` as a copy of the code in lines 27, 33 and 39 would make maintenance of this class more difficult. If the implementations of `setHour`, `setMinute` and `setSecond` were to change, the implementation of any member function that duplicates lines 27, 33 and 39 would have to change accordingly. Having the `Time` constructor call `setTime` and having `setTime` call `setHour`, `setMinute` and `setSecond` enables us to limit the changes to code that validates the `hour`, `minute` or `second` to the corresponding `set` function. This reduces the likelihood of errors when altering the class's implementation. Also, the performance of the `Time` constructor and `setTime` can be enhanced by explicitly declaring them `inline` or by defining them in the class definition (which implicitly inlines the function definition).

**Software Engineering Observation 17.9**

If a member function of a class already provides all or part of the functionality required by a constructor (or other member function) of the class, call that member function from the constructor (or other member function). This simplifies the maintenance of the code and reduces the likelihood of an error if the implementation of the code is modified. As a general rule: Avoid repeating code.

**Software Engineering Observation 17.10**

Any change to the default argument values of a function requires the client code to be recompiled (to ensure that the program still functions correctly).

Function `main` in Fig. 17.10 initializes five `Time` objects—one with all three arguments defaulted in the implicit constructor call (line 9), one with one argument specified (line 10), one with two arguments specified (line 11), one with three arguments specified (line 12) and one with three invalid arguments specified (line 13). Then the program displays each object in universal-time and standard-time formats.

```cpp
// Fig. 17.10: fig17_10.cpp
// Demonstrating a default constructor for class Time.
#include <iostream>
#include "Time.h" // include definition of class Time from Time.h
using namespace std;

int main()
{
    Time t1; // all arguments defaulted
    Time t2( 2 ); // hour specified; minute and second defaulted
    Time t3( 21, 34 ); // hour and minute specified; second defaulted
    Time t4( 12, 25, 42 ); // hour, minute and second specified
    Time t5( 27, 74, 99 ); // all bad values specified
    cout << "Constructed with:\n\n\nt1: all arguments defaulted\n  ";
t1.printUniversal(); // 00:00:00
    cout << "\n  ";
```

**Fig. 17.10** Constructor with default arguments. (Part I of 2.)
Notes Regarding Class Time’s Set and Get Functions and Constructor

Time’s set and get functions are called throughout the class’s body. In particular, function setTime (lines 17–22 of Fig. 17.9) calls functions setHour, setMinute and setSecond, and functions printUniversal and printStandard call functions getHour, getMinute and getSecond in line 63–64 and lines 70–72, respectively. In each case, these functions could have accessed the class’s private data directly. However, consider changing the rep-
presentation of the time from three int values (requiring 12 bytes of memory) to a single
int value representing the total number of seconds that have elapsed since midnight (re-
quiring only four bytes of memory). If we made such a change, only the bodies of the func-
tions that access the private data directly would need to change—in particular, the
individual set and get functions for the hour, minute and second. There would be no need
to modify the bodies of functions setTime, printUniversal or printStandard, because they
do not access the data directly. Designing the class in this manner reduces the likeli-
hood of programming errors when altering the class’s implementation.

Similarly, the Time constructor could be written to include a copy of the appropriate
statements from function setTime. Doing so may be slightly more efficient, because the
extra constructor call and call to setTime are eliminated. However, duplicating statements
in multiple functions or constructors makes changing the class’s internal data representa-
tion more difficult. Having the Time constructor call function setTime directly requires
any changes to the implementation of setTime to be made only once.

**Common Programming Error 17.2**

A constructor can call other member functions of the class, such as set or get functions, but
because the constructor is initializing the object, the data members may not yet be in a
consistent state. Using data members before they have been properly initialized can cause
logic errors.

### 17.7 Destructors

A destructor is another type of special member function. The name of the destructor for
a class is the tilde character (~) followed by the class name. This naming convention has
intuitive appeal, because as we’ll see in a later chapter, the tilde operator is the bitwise com-
plement operator, and, in a sense, the destructor is the complement of the constructor. A
destructor is often referred to with the abbreviation “dtor” in the literature. We prefer not
to use this abbreviation.

A class’s destructor is called implicitly when an object is destroyed. This occurs, for
example, as an automatic object is destroyed when program execution leaves the scope in
which that object was instantiated. The destructor itself does not actually release the object’s
memory—it performs termination housekeeping before the object’s memory is reclaimed,
so the memory may be reused to hold new objects.

A destructor receives no parameters and returns no value. A destructor may not
specify a return type—not even void. A class may have only one destructor—destructor
overloading is not allowed. A destructor must be public.

**Common Programming Error 17.3**

It’s a syntax error to attempt to pass arguments to a destructor, to specify a return type for
a destructor (even void cannot be specified), to return values from a destructor or to over-
load a destructor.

Even though destructors have not been provided for the classes presented so far, every
class has a destructor. If you do not explicitly provide a destructor, the compiler creates an
“empty” destructor. [Note: We’ll see that such an implicitly created destructor does, in
fact, perform important operations on objects that are created through composition
(Chapter 18) and inheritance (Chapter 20).] In Chapter 19, we’ll build destructors appropriate for classes whose objects contain dynamically allocated memory (e.g., for arrays and strings) or use other system resources (e.g., files on disk). We discuss how to dynamically allocate and deallocate memory in Chapter 18.

Software Engineering Observation 17.11

Constructors and destructors have much greater prominence in C++ and object-oriented programming than is possible to convey after only our brief introduction here.

17.8 When Constructors and Destructors are Called

Constructors and destructors are called implicitly by the compiler. The order in which these function calls occur depends on the order in which execution enters and leaves the scopes where the objects are instantiated. Generally, destructor calls are made in the reverse order of the corresponding constructor calls, but as we’ll see in Figs. 17.11–17.13, the storage classes of objects can alter the order in which destructors are called.

Constructors are called for objects defined in global scope before any other function (including main) in that file begins execution (although the order of execution of global object constructors between files is not guaranteed). The corresponding destructors are called when main terminates. Function exit forces a program to terminate immediately and does not execute the destructors of automatic objects. The function often is used to terminate a program when an error is detected in the input or if a file to be processed by the program cannot be opened. Function abort performs similarly to function exit but forces the program to terminate immediately, without allowing the destructors of any objects to be called. Function abort is usually used to indicate an abnormal termination of the program. (See Chapter 14, for more information on functions exit and abort.)

The constructor for an automatic local object is called when execution reaches the point where that object is defined—the corresponding destructor is called when execution leaves the object’s scope (i.e., the block in which that object is defined has finished executing). Constructors and destructors for automatic objects are called each time execution enters and leaves the scope of the object. Destructors are not called for automatic objects if the program terminates with a call to function exit or function abort.

The constructor for a static local object is called only once, when execution first reaches the point where the object is defined—the corresponding destructor is called when main terminates or the program calls function exit. Global and static objects are destroyed in the reverse order of their creation. Destructors are not called for static objects if the program terminates with a call to function abort.

The program of Figs. 17.11–17.13 demonstrates the order in which constructors and destructors are called for objects of class CreateAndDestroy (Fig. 17.11 and Fig. 17.12) of various storage classes in several scopes. Each object of class CreateAndDestroy contains an integer (objectID) and a string (message) that are used in the program’s output to identify the object (Fig. 17.11 lines 16–17). This mechanical example is purely for pedagogic purposes. For this reason, line 21 of the destructor in Fig. 17.12 determines whether the object being destroyed has an objectID value 1 or 6 and, if so, outputs a newline character. This line makes the program’s output easier to follow.
Chapter 17  Classes: A Deeper Look, Part 1

Figure 17.13 defines object first (line 10) in global scope. Its constructor is actually called before any statements in main execute and its destructor is called at program termination after the destructors for all other objects have run.

```cpp
// Fig. 17.11: CreateAndDestroy.h
// CreateAndDestroy class definition.
// Member functions defined in CreateAndDestroy.cpp.
#include <string>
using namespace std;

#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy
{
public:
    CreateAndDestroy( int, string ); // constructor
    ~CreateAndDestroy(); // destructor

private:
    int objectID; // ID number for object
    string message; // message describing object
}; // end class CreateAndDestroy
#endif
```

**Fig. 17.11**  |  CreateAndDestroy class definition.

```cpp
// Fig. 17.12: CreateAndDestroy.cpp
// CreateAndDestroy class member-function definitions.
#include <iostream>
#include "CreateAndDestroy.h" // include CreateAndDestroy class definition
using namespace std;

CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
{
    objectID = ID; // set object's ID number
    message = messageString; // set object's descriptive message
    cout << "Object " << objectID << " constructor runs " << message << endl;
} // end CreateAndDestroy constructor

CreateAndDestroy::~CreateAndDestroy()
{
    // output newline for certain objects; helps readability
    cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );
    cout << "Object " << objectID << " destructor runs " << message << endl;
} // end ~CreateAndDestroy destructor
```

**Fig. 17.12**  |  CreateAndDestroy class member-function definitions.

Figure 17.13 defines object first (line 10) in global scope. Its constructor is actually called before any statements in main execute and its destructor is called at program termination after the destructors for all other objects have run.
Function main (lines 12–23) declares three objects. Objects second (line 15) and fourth (line 21) are local automatic objects, and object third (line 16) is a static local object. The constructor for each of these objects is called when execution reaches the point where that object is declared. The destructors for objects fourth then second are called (i.e., the reverse of the order in which their constructors were called) when execution reaches the end of main. Because object third is static, it exists until program termination. The destructor for object third is called before the destructor for global object first, but after all other objects are destroyed.

Function create (lines 26–33) declares three objects—fifth (line 29) and seventh (line 31) as local automatic objects, and sixth (line 30) as a static local object. The destructors for objects seventh then fifth are called (i.e., the reverse of the order in which their constructors were called) when create terminates. Because sixth is static, it exists until program termination. The destructor for sixth is called before the destructors for third and first, but after all other objects are destroyed.

---

```cpp
// Fig. 17.13: fig17_13.cpp
// Demonstrating the order in which constructors and destructors are called.
#include <iostream>
#include "CreateAndDestroy.h" // include CreateAndDestroy class definition
using namespace std;

void create( void ); // prototype

CreateAndDestroy first( 1, "(global before main)" ); // global object

int main()
{
    cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
    CreateAndDestroy second( 2, "(local automatic in main)" );
    static CreateAndDestroy third( 3, "(local static in main)" );
    create(); // call function to create objects
    cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
    CreateAndDestroy fourth( 4, "(local automatic in main)" );
    cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
} // end main

// function to create objects
void create( void )
{
    cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
    CreateAndDestroy fifth( 5, "(local automatic in create)" );
    static CreateAndDestroy sixth( 6, "(local static in create)" );
    CreateAndDestroy seventh( 7, "(local automatic in create)" );
    cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
} // end function create
```

---

Fig. 17.13 | Order in which constructors and destructors are called. (Part 1 of 2.)
17.9 Time Class Case Study: A Subtle Trap—Returning a Reference to a private Data Member

A reference to an object is an alias for the name of the object and, hence, may be used on the left side of an assignment statement. In this context, the reference makes a perfectly acceptable lvalue that can receive a value. One way to use this capability (unfortunately!) is to have a public member function of a class return a reference to a private data member of that class. If a function returns a const reference, that reference cannot be used as a modifiable lvalue.

The program of Figs. 17.14–17.16 uses a simplified Time class (Fig. 17.14 and Fig. 17.15) to demonstrate returning a reference to a private data member with member function badSetHour (declared in Fig. 17.14 in line 15 and defined in Fig. 17.15 in lines 27–31). Such a reference return actually makes a call to member function badSetHour an alias for private data member hour! The function call can be used in any way that the private data member can be used, including as an lvalue in an assignment statement, thus enabling clients of the class to clobber the class’s private data at will! The same problem would occur if a pointer to the private data were to be returned by the function.
A Subtle Trap—Returning a Reference to a private Data Member

```cpp
// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H

class Time
{
    public:
    Time( int = 0, int = 0, int = 0 );
    void setTime( int, int, int );
    int getHour();
    int &badSetHour( int ); // DANGEROUS reference return

    private:
    int hour;
    int minute;
    int second;
}; // end class Time

#endif
```

**Fig. 17.14** | Time class declaration. (Part 2 of 2.)

```cpp
// Fig. 17.15: Time.cpp
// Time class member-function definitions.
#include "Time.h" // include definition of class Time

// constructor function to initialize private data; calls member function
// setTime to set variables; default values are 0 (see class definition)
Time::Time( int hr, int min, int sec )
{
    setTime( hr, min, sec );
} // end Time constructor

// set values of hour, minute and second
void Time::setTime( int h, int m, int s )
{
    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setTime

// return hour value
int Time::getHour()
{
    return hour;
} // end function getHour

// POOR PRACTICE: Returning a reference to a private data member.
int &Time::badSetHour( int hh )
{
    hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
    return hour; // DANGEROUS reference return
} // end function badSetHour
```

**Fig. 17.15** | Time class member-function definitions.
Figure 17.16 declares Time object t (line 10) and reference hourRef (line 13), which is initialized with the reference returned by the call t.badSetHour(20). Line 15 displays the value of the alias hourRef. This shows how hourRef breaks the encapsulation of the class—statements in main should not have access to the private data of the class. Next, line 16 uses the alias to set the value of hour to 30 (an invalid value) and line 17 displays the value returned by function getHour to show that assigning a value to hourRef actually modifies the private data in the Time object t. Finally, line 21 uses the badSetHour function call itself as an lvalue and assigns 74 (another invalid value) to the reference returned by the function. Line 26 again displays the value returned by function getHour to show that assigning a value to the result of the function call in line 21 modifies the private data in the Time object t.

```
// Fig. 17.16: fig17_16.cpp
// Demonstrating a public member function that
// returns a reference to a private data member.
#include <iostream>
#include "Time.h" // include definition of class Time
using namespace std;

int main()
{
    Time t; // create Time object
    int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour

    cout << "Valid hour before modification: " << hourRef;
    hourRef = 30; // use hourRef to set invalid value in Time object t
    cout << "Invalid hour after modification: " << t.getHour();

    // Dangerous: Function call that returns
    // a reference can be used as an lvalue!
    t.badSetHour( 12 ) = 74; // assign another invalid value to hour

    cout << "\n\nPOOR PROGRAMMING PRACTICE!!!!!!!!\n"
    << "t.badSetHour( 12 ) as an lvalue, invalid hour: "
    << t.getHour();
    cout << "\n*******************************************************************\n" << endl;
} // end main
```

Valid hour before modification: 20
Invalid hour after modification: 30

POOR PROGRAMMING PRACTICE!!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74

Fig. 17.16  |  Returning a reference to a private data member.
17.10 Default Memberwise Assignment

The assignment operator (=) can be used to assign an object to another object of the same type. By default, such assignment is performed by memberwise assignment—each data member of the object on the right of the assignment operator is assigned individually to the same data member in the object on the left of the assignment operator.

Figures 17.17–17.18 define class Date for use in this example. Line 18 of Fig. 17.19 uses default memberwise assignment to assign the data members of Date object date1 to the corresponding data members of Date object date2. In this case, the month member of date1 is assigned to the month member of date2, the day member of date1 is assigned to the day member of date2 and the year member of date1 is assigned to the year member of date2. [Caution: Memberwise assignment can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory; we discuss these problems in Chapter 19 and show how to deal with them.] The Date constructor does not contain any error checking; we leave this to the exercises.

```
// Fig. 17.17: Date.h
// Date class declaration. Member functions are defined in Date.cpp.

// prevent multiple inclusions of header file
#ifndef DATE_H
#define DATE_H

// class Date definition
class Date
{
  public:
    Date( int = 1, int = 1, int = 2000 ); // default constructor
    void print();
  private:
    int month;
    int day;
    int year;
}; // end class Date
#endif
```

Fig. 17.17 | Date class declaration.

```
// Fig. 17.18: Date.cpp
// Date class member-function definitions.
#include <iostream>
#include "Date.h" // include definition of class Date from Date.h
```

Fig. 17.18 | Date class member-function definitions. (Part 1 of 2.)
Objects may be passed as function arguments and may be returned from functions. Such passing and returning is performed using pass-by-value by default—a copy of the object is passed or returned. In such cases, C++ creates a new object and uses a copy constructor to copy the original object’s values into the new object. For each class, the compiler
provides a default copy constructor that copies each member of the original object into the corresponding member of the new object. Like memberwise assignment, copy constructors can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory. Chapter 19 discusses how to define customized copy constructors that properly copy objects containing pointers to dynamically allocated memory.

**Performance Tip 17.3**

Passing an object by value is good from a security standpoint, because the called function has no access to the original object in the caller, but pass-by-value can degrade performance when making a copy of a large object. An object can be passed by reference by passing either a pointer or a reference to the object. Pass-by-reference offers good performance but is weaker from a security standpoint, because the called function is given access to the original object. Pass-by-const-reference is a safe, good-performing alternative (this can be implemented with a const reference parameter or with a pointer-to-const-data parameter).

### 17.11 Wrap-Up

This chapter deepened our coverage of classes, using a rich `Time` class case study to introduce several new features. You saw that member functions are usually shorter than global functions because member functions can directly access an object’s data members, so the member functions can receive fewer arguments than functions in procedural programming languages. You learned how to use the arrow operator to access an object’s members via a pointer of the object’s class type.

You learned that member functions have class scope—the member function’s name is known only to the class’s other members unless referred to via an object of the class, a reference to an object of the class, a pointer to an object of the class or the binary scope resolution operator. We also discussed access functions (commonly used to retrieve the values of data members or to test the truth or falsity of conditions) and utility functions (private member functions that support the operation of the class’s public member functions).

You learned that a constructor can specify default arguments that enable it to be called in a variety of ways. You also learned that any constructor that can be called with no arguments is a default constructor and that there can be at most one default constructor per class. We discussed destructors and their purpose of performing termination housekeeping on an object of a class before that object is destroyed. We also demonstrated the order in which an object’s constructors and destructors are called.

We demonstrated the problems that can occur when a member function returns a reference to a private data member, which breaks the encapsulation of the class. We also showed that objects of the same type can be assigned to one another using default memberwise assignment. We also discussed the benefits of using class libraries to enhance the speed with which code can be created and to increase the quality of software.

Chapter 18 presents additional class features. We’ll demonstrate how `const` can be used to indicate that a member function does not modify an object of a class. You’ll build classes with composition, which allows a class to contain objects of other classes as members. We’ll show how a class can allow so-called “friend” functions to access the class’s non-public members. We’ll also show how a class’s non-static member functions can use a special pointer named `this` to access an object’s members.
Summary

Section 17.2 Time Class Case Study
- Preprocessor directives `#ifndef` (which means “if not defined”) and `#endif` are used to prevent multiple inclusions of a header file. If the code between these directives has not previously been included in an application, `#define` defines a name that can be used to prevent future inclusions, and the code is included in the source code file.
- Data members cannot be initialized where they’re declared in the class body (except for a class’s `static const` data members of integral or `enum` types). Initialize these data members in the class’s constructor (as there is no default initialization for data members of fundamental types).
- Stream manipulator `setfill` specifies the fill character that is displayed when an integer is output in a field that is wider than the number of digits in the value.
- By default, the fill characters appear before the digits in the number.
- Stream manipulator `setfill` is a “sticky” setting, meaning that once the fill character is set, it applies for all subsequent fields being printed.
- Even though a member function declared in a class definition may be defined outside that class definition (and “tied” to the class via the binary scope resolution operator), that member function is still within that class’s scope.
- If a member function is defined in the body of a class definition, the C++ compiler attempts to inline calls to the member function.
- Classes can include objects of other classes as members or they may be derived from other classes that provide attributes and behaviors the new classes can use.

Section 17.3 Class Scope and Accessing Class Members
- A class’s data members and member functions belong to that class’s scope.
- Nonmember functions are defined at global namespace scope.
- Within a class’s scope, class members are immediately accessible by all of that class’s member functions and can be referenced by name.
- Outside a class’s scope, class members are referenced through one of the handles on an object—an object name, a reference to an object or a pointer to an object.
- Member functions of a class can be overloaded, but only by other member functions of that class.
- To overload a member function, provide in the class definition a prototype for each version of the overloaded function, and provide a separate definition for each version of the function.
- Variables declared in a member function have local scope and are known only to that function.
- If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the block-scope variable in the local scope.
- The dot member selection operator (.) is preceded by an object’s name or by a reference to an object to access the object’s public members.
- The arrow member selection operator (`->`) is preceded by a pointer to an object to access that object’s public members.

Section 17.4 Separating Interface from Implementation
- Header files contain some portions of a class’s implementation and hints about others. Inline member functions, for example, should be in a header file, so that when the compiler compiles a client, the client can include the inline function definition in place.
- A class’s private members that are listed in the class definition in the header file are visible to clients, even though the clients may not access the private members.
Section 17.5 Access Functions and Utility Functions
• A utility function is a private member function that supports the operation of the class’s public member functions. Utility functions are not intended to be used by clients of a class.

Section 17.6 Time Class Case Study: Constructors with Default Arguments
• Like other functions, constructors can specify default arguments.

Section 17.7 Destructors
• A class’s destructor is called implicitly when an object of the class is destroyed.
• The name of the destructor for a class is the tilde (–) character followed by the class name.
• A destructor does not release an object’s storage—it performs termination housekeeping before the system reclaims an object’s memory, so the memory may be reused to hold new objects.
• A destructor receives no parameters and returns no value. A class may have only one destructor.
• If you do not explicitly provide a destructor, the compiler creates an “empty” destructor, so every class has exactly one destructor.

Section 17.8 When Constructors and Destructors are Called
• The order in which constructors and destructors are called depends on the order in which execution enters and leaves the scopes where the objects are instantiated.
• Generally, destructor calls are made in the reverse order of the corresponding constructor calls, but the storage classes of objects can alter the order in which destructors are called.

Section 17.9 Time Class Case Study: A Subtle Trap—Returning a Reference to a private Data Member
• A reference to an object is an alias for the name of the object and, hence, may be used on the left side of an assignment statement. In this context, the reference makes a perfectly acceptable lvalue that can receive a value.
• If the function returns a const reference, then the reference cannot be used as a modifiable lvalue.

Section 17.10 Default Memberwise Assignment
• The assignment operator (=) can be used to assign an object to another object of the same type. By default, such assignment is performed by memberwise assignment.
• Objects may be passed by value to or returned by value from functions. C++ creates a new object and uses a copy constructor to copy the original object’s values into the new object.
• For each class, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object.

Terminology
abort function 621
destructor 620

access function 612
#define preprocessor directive 604

aggregation 609
exit function 621

arrow member selection operator (–>) 610
fill character 606

class scope 606
handle on an object 609

composition 609
helper function 612

copy constructor 628
#include preprocessor directive 604

default memberwise assignment 627
inheritance 609

define preprocessor directive 604
memberwise assignment 627

derive one class from another 609
predicate function 612

derived 609
preprocessor wrapper 604
Self-Review Exercises

17.1 Fill in the blanks in each of the following:
   a) Class members are accessed via the _____ operator in conjunction with the name of an object (or reference to an object) of the class or via the _____ operator in conjunction with a pointer to an object of the class.
   b) Class members specified as _______ are accessible only to member functions of the class and friends of the class.
   c) Class members specified as _______ are accessible anywhere an object of the class is in scope.
   d) _______ can be used to assign an object of a class to another object of the same class.

17.2 Find the error(s) in each of the following and explain how to correct it (them):
   a) Assume the following prototype is declared in class Time:
      ```cpp
      void ~Time( int );
      ```
   b) The following is a partial definition of class Time:
      ```cpp
      class Time
      {
      public:
          // function prototypes
      private:
          int hour = 0;
          int minute = 0;
          int second = 0;
      }; // end class Time
      ```
   c) Assume the following prototype is declared in class Employee:
      ```cpp
      int Employee( string, string );
      ```

Answers to Self-Review Exercises

17.1 a) dot (.), arrow (->). b) private. c) public. d) Default memberwise assignment (performed by the assignment operator).

17.2 a) Error: Destructors are not allowed to return values (or even specify a return type) or take arguments.
   Correction: Remove the return type void and the parameter int from the declaration.
   b) Error: Members cannot be explicitly initialized in the class definition.
   Correction: Remove the explicit initialization from the class definition and initialize the data members in a constructor.
   c) Error: Constructors are not allowed to return values.
   Correction: Remove the return type int from the declaration.

Exercises

17.3 What’s the purpose of the scope resolution operator?

17.4 (Enhancing Class Time) Provide a constructor that is capable of using the current time from the time and localtime functions—declared in the C++ Standard Library header <ctime>—to initialize an object of the Time class.
17.5  (*Complex Class*) Create a class called *Complex* for performing arithmetic with complex numbers. Write a program to test your class. Complex numbers have the form
\[ \text{realPart} + \text{imaginaryPart} \times i \]
where \( i \) is \( \sqrt{-1} \)

Use *double* variables to represent the private data of the class. Provide a constructor that enables an object of this class to be initialized when it’s declared. The constructor should contain default values in case no initializers are provided. Provide public member functions that perform the following tasks:

a) Adding two *Complex* numbers: The real parts are added together and the imaginary parts are added together.

b) Subtracting two *Complex* numbers: The real part of the right operand is subtracted from the real part of the left operand, and the imaginary part of the right operand is subtracted from the imaginary part of the left operand.

c) Printing *Complex* numbers in the form \((a, b)\), where \( a \) is the real part and \( b \) is the imaginary part.

17.6  (*Rational Class*) Create a class called *Rational* for performing arithmetic with fractions. Write a program to test your class.

Use integer variables to represent the private data of the class—the numerator and the denominator. Provide a constructor that enables an object of this class to be initialized when it’s declared. The constructor should contain default values in case no initializers are provided and should store the fraction in reduced form. For example, the fraction \( \frac{2}{4} \) would be stored in the object as 1 in the numerator and 2 in the denominator. Provide public member functions that perform each of the following tasks:

a) Adding two *Rational* numbers. The result should be stored in reduced form.

b) Subtracting two *Rational* numbers. The result should be stored in reduced form.

c) Multiplying two *Rational* numbers. The result should be stored in reduced form.

d) Dividing two *Rational* numbers. The result should be stored in reduced form.

e) Printing *Rational* numbers in the form \( \frac{a}{b} \), where \( a \) is the numerator and \( b \) is the denominator.

f) Printing *Rational* numbers in floating-point format.

17.7  (*Enhancing Class Time*) Modify the *Time* class of Figs. 17.8–17.9 to include a *tick* member function that increments the time stored in a *Time* object by one second. The *Time* object should always remain in a consistent state. Write a program that tests the *tick* member function in a loop that prints the time in standard format during each iteration of the loop to illustrate that the *tick* member function works correctly. Be sure to test the following cases:

a) Incrementing into the next minute.

b) Incrementing into the next hour.

c) Incrementing into the next day (i.e., 11:59:59 PM to 12:00:00 AM).

17.8  (*Enhancing Class Date*) Modify the *Date* class of Figs. 17.17–17.18 to include error checking on the initializer values for data members *month*, *day* and *year*. Also, provide a member function *nextDay* to increment the day by one. The *Date* object should always remain in a consistent state. Write a program that tests function *nextDay* in a loop that prints the date in standard format during each iteration of the loop to illustrate that *nextDay* works correctly. Be sure to test the following cases:

a) Incrementing into the next month.

b) Incrementing into the next year.
17.9 (Combining Class Time and Class Date) Combine the modified Time class of Exercise 17.7 and the modified Date class of Exercise 17.8 into one class called DateAndTime. (In Chapter 20, we’ll discuss inheritance, which will enable us to accomplish this task quickly without modifying the existing class definitions.) Modify the tick function to call the nextDay function if the time increments into the next day. Modify functions printStandard and printUniversal to output the date and time. Write a program to test the new class DateAndTime. Specifically, test incrementing the time into the next day.

17.10 (Returning Error Indicators from Class Time’s set Functions) Modify the set functions in the Time class of Figs. 17.8–17.9 to return appropriate error values if an attempt is made to set a data member of an object of class Time to an invalid value. Write a program that tests your new version of class Time. Display error messages when set functions return error values.

17.11 (Rectangle Class) Create a class Rectangle with attributes length and width, each of which defaults to 1. Provide member functions that calculate the perimeter and the area of the rectangle. Also, provide set and get functions for the length and width attributes. The set functions should verify that length and width are each floating-point numbers larger than 0.0 and less than 20.0.

17.12 (Enhancing Class Rectangle) Create a more sophisticated Rectangle class than the one you created in Exercise 17.11. This class stores only the Cartesian coordinates of the four corners of the rectangle. The constructor calls a set function that accepts four sets of coordinates and verifies that each of these is in the first quadrant with no single x- or y-coordinate larger than 20.0. The set function also verifies that the supplied coordinates do, in fact, specify a rectangle. Provide member functions that calculate the length, width, perimeter, and area. The length is the larger of the two dimensions. Include a predicate function square that determines whether the rectangle is a square.

17.13 (Enhancing Class Rectangle) Modify class Rectangle from Exercise 17.12 to include a draw function that displays the rectangle inside a 25-by-25 box enclosing the portion of the first quadrant in which the rectangle resides. Include a setFillColor function to specify the character out of which the body of the rectangle will be drawn. Include a setPerimeterCharacter function to specify the character that will be used to draw the border of the rectangle. If you feel ambitious, you might include functions to scale the size of the rectangle, rotate it, and move it around within the designated portion of the first quadrant.

17.14 (HugeInteger Class) Create a class HugeInteger that uses a 40-element array of digits to store integers as large as 40 digits each. Provide member functions input, output, add and subtract. For comparing HugeInteger objects, provide functions isEqualTo, isNotEqualTo, isGreaterThan, isLessThan, isGreaterThanOrEqualTo and isLessThanOrEqualTo—each of these is a “predicate” function that simply returns true if the relationship holds between the two HugeIntegers and returns false if the relationship does not hold. Also, provide a predicate function isZero. If you feel ambitious, provide member functions multiply, divide and modulus.

17.15 (TicTacToe Class) Create a class TicTacToe that will enable you to write a complete program to play the game of tic-tac-toe. The class contains as private data a 3-by-3 two-dimensional array of integers. The constructor should initialize the empty board to all zeros. Allow two human players. Wherever the first player moves, place a 1 in the specified square. Place a 2 wherever the second player moves. Each move must be to an empty square. After each move, determine whether the game has been won or is a draw. If you feel ambitious, modify your program so that the computer makes the moves for one of the players. Also, allow the player to specify whether he or she wants to go first or second. If you feel exceptionally ambitious, develop a program that will play three-dimensional tic-tac-toe on a 4-by-4-by-4 board. [Caution: This is an extremely challenging project that could take many weeks of effort!]

Objectives
In this chapter you’ll learn:

- To specify `const` (constant) objects and `const` member functions.
- To create objects composed of other objects.
- To use `friend` functions and `friend` classes.
- To use the `this` pointer.
- To use `static` data members and member functions.
- The concept of a container class.
- The notion of iterator classes that walk through the elements of container classes.
- To use proxy classes to hide implementation details from a class’s clients.
18.1 Introduction

In this chapter, we continue our study of classes and data abstraction with several more advanced topics. We use const objects and const member functions to prevent modifications of objects and enforce the principle of least privilege. We discuss composition—a form of reuse in which a class can have objects of other classes as members. Next, we introduce friendship, which enables a class designer to specify nonmember functions that can access a class’s non-public members—a technique that is often used in operator overloading (Chapter 19) for performance reasons. We discuss a special pointer (called this), which is an implicit argument to each of a class’s non-static member functions. It allows those member functions to access the correct object’s data members and other non-static member functions. Finally, we motivate the need for static class members and show how to use static data members and member functions in your own classes.

18.2 const (Constant) Objects and const Member Functions

Let’s see how the principle of least privilege applies to objects. Some objects need to be modifiable and some do not. You may use keyword const to specify that an object is not modifiable and that any attempt to modify the object should result in a compilation error. The statement

```cpp
const Time noon(12, 0, 0);
```

declares a const object noon of class Time and initializes it to 12 noon.

Software Engineering Observation 18.1

Attempts to modify a const object are caught at compile time rather than causing execution-time errors.

Performance Tip 18.1

Declaring variables and objects const when appropriate can improve performance—compilers can perform certain optimizations on constants that cannot be performed on variables.
C++ disallows member function calls for const objects unless the member functions themselves are also declared const. This is true even for get member functions that do not modify the object.

A member function is specified as const both in its prototype (Fig. 18.1; lines 19–24) and in its definition (Fig. 18.2; lines 43, 49, 55 and 61) by inserting the keyword const after the function's parameter list and, in the case of the function definition, before the left brace that begins the function body.

Common Programming Error 18.1
Defining as const a member function that modifies a data member of the object is a compilation error.

Common Programming Error 18.2
Defining as const a member function that calls a non-const member function of the class on the same object is a compilation error.

Common Programming Error 18.3
Invoking a non-const member function on a const object is a compilation error.

Software Engineering Observation 18.2
A const member function can be overloaded with a non-const version. The compiler chooses which overloaded member function to use based on the object on which the function is invoked. If the object is const, the compiler uses the const version. If the object is not const, the compiler uses the non-const version.

An interesting problem arises for constructors and destructors, each of which typically modifies objects. A constructor must be allowed to modify an object so that the object can be initialized properly. A destructor must be able to perform its termination housekeeping chores before an object’s memory is reclaimed by the system.

Common Programming Error 18.4
Attempting to declare a constructor or destructor const is a compilation error.

Defining and Using const Member Functions
The program of Figs. 18.1–18.3 modifies class Time of Figs. 17.8–17.9 by making its get functions and printUniversal function const. In the header file Time.h (Fig. 18.1), lines 19–21 and 24 now include keyword const after each function’s parameter list. The corresponding definition of each function in Fig. 18.2 (lines 43, 49, 55 and 61, respectively) also specifies keyword const after each function’s parameter list.

```
// Fig. 18.1: Time.h
// Time class definition with const member functions.
// Member functions defined in Time.cpp.
#ifndef TIME_H
#define TIME_H

1 // Fig. 18.1: Time.h
2 // Time class definition with const member functions.
3 // Member functions defined in Time.cpp.
4 #ifndef TIME_H
5 #define TIME_H
```
```cpp
class Time {
public:
    Time(int = 0, int = 0, int = 0); // default constructor

    // set functions
    void setTime(int, int, int); // set time
    void setHour(int); // set hour
    void setMinute(int); // set minute
    void setSecond(int); // set second

    // get functions (normally declared const)
    int getHour() const; // return hour
    int getMinute() const; // return minute
    int getSecond() const; // return second

    // print functions (normally declared const)
    void printUniversal() const; // print universal time
    void printStandard(); // print standard time (should be const)

private:
    int hour; // 0 - 23 (24-hour clock format)
    int minute; // 0 - 59
    int second; // 0 - 59
}; // end class Time
```

---

**Fig. 18.1** | Time class definition with const member functions. (Part 2 of 2.)

---

```cpp
// Fig. 18.2: Time.cpp
// Time class member-function definitions.
#include <iostream>
#include <iomanip>
#include "Time.h" // include definition of class Time
using namespace std;

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time(int hour, int minute, int second) { 
    setTime(hour, minute, second); // end Time constructor
}

// set hour, minute and second values
void Time::setTime(int hour, int minute, int second) {
    setHour(hour);
    setMinute(minute);
    setSecond(second);
} // end function setTime
```

---

**Fig. 18.2** | Time class member-function definitions. (Part 1 of 2.)
Fig. 18.2 | Time class member-function definitions. (Part 2 of 2.)
Chapter 18  Classes: A Deeper Look, Part 2

Figure 18.3 instantiates two Time objects—non-const object wakeUp (line 7) and const object noon (line 8). The program attempts to invoke non-const member functions setHour (line 13) and printStandard (line 20) on the const object noon. In each case, the compiler generates an error message. The program also illustrates the three other member-function-call combinations on objects—a non-const member function on a non-const object (line 11), a const member function on a non-const object (line 15) and a const member function on a const object (lines 17–18). The error messages generated for non-const member functions called on a const object are shown in the output window.

```cpp
// Fig. 18.3: fig18_03.cpp
// Attempting to access a const object with non-const member functions.
#include "Time.h"  // include Time class definition

int main()
{
    Time wakeUp(6, 45, 0);  // non-constant object
    const Time noon(12, 0, 0); // constant object

    wakeUp.setHour(18);    // non-const  non-const
    noon.setHour(12);      // const      non-const
    wakeUp.getHour();      // non-const  const
    noon.getMinute();      // const      const
    noon.printUniversal(); // const      const
    noon.printStandard();  // const      non-const
}
```

Microsoft Visual C++ compiler error messages:

```
C:\examples\ch18\Fig18_01_03\fig18_03.cpp(13) : error C2662:
'Time::setHour' : cannot convert 'this' pointer from 'const Time' to 'Time &'
Conversion loses qualifiers
C:\examples\ch18\Fig18_01_03\fig18_03.cpp(20) : error C2662:
'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to 'Time &'
Conversion loses qualifiers
```

GNU C++ compiler error messages:

```
fig18_03.cpp:13: error: passing 'const Time' as 'this' argument of
    'void Time::setHour(int)' discards qualifiers
fig18_03.cpp:20: error: passing 'const Time' as 'this' argument of
    'void Time::printStandard()' discards qualifiers
```

Fig. 18.3  const objects and const member functions.
A constructor must be a non-const member function (Fig. 18.2, lines 11–14), but it can still be used to initialize a const object (Fig. 18.3, line 8). The Time constructor’s definition (Fig. 18.2, lines 11–14) shows that it calls another non-const member function—setTime (lines 17–22)—to perform the initialization of a Time object. Invoking a non-const member function from the constructor call as part of the initialization of a const object is allowed. The “constness” of a const object is enforced from the time the constructor completes initialization of the object until that object’s destructor is called.

Also, line 20 in Fig. 18.3 generates a compilation error even though member function printStandard of class Time does not modify the object on which it’s invoked. The fact that a member function does not modify an object is not sufficient to indicate that the function is constant function—the function must explicitly be declared const.

**Initializing a const Data Member with a Member Initializer**

The program of Figs. 18.4–18.6 introduces using member initializer syntax. All data members can be initialized using member initializer syntax, but const data members and data members that are references must be initialized using member initializers. Later in this chapter, we’ll see that member objects must be initialized this way as well.

```
1 // Fig. 18.4: Increment.h
2 // Definition of class Increment.
3 #ifndef INCREMENT_H
4 #define INCREMENT_H
5
6 class Increment
7 {
8    public:
9        Increment( int c = 0, int i = 1 ); // default constructor
10    
11        // function addIncrement definition
12        void addIncrement()
13        {
14            count += increment;
15        } // end function addIncrement
16
17        void print() const; // prints count and increment
18    private:
19        int count;
20        const int increment; // const data member
21    }; // end class Increment
22
23 #endif
```

**Fig. 18.4** | Increment class definition containing non-const data member count and const data member increment.

```
1 // Fig. 18.5: Increment.cpp
2 // Member-function definitions for class Increment demonstrate using a
3 // member initializer to initialize a constant of a built-in data type.
```

**Fig. 18.5** | Member initializer used to initialize a constant of a built-in data type. (Part 1 of 2.)
Chapter 18 Classes: A Deeper Look, Part 2

The constructor definition (Fig. 18.5, lines 9–14) uses a **member initializer list** to initialize class `Increment`’s data members—non-const integer `count` and const integer `increment` (declared in lines 19–20 of Fig. 18.4). Member initializers appear between a constructor’s parameter list and the left brace that begins the constructor’s body. The

```cpp
#include <iostream>
#include "Increment.h" // include definition of class Increment
using namespace std;

// constructor
Increment::Increment( int c, int i )
    : count( c ), // initializer for non-const member
      increment( i ) // required initializer for const member
{
    // empty body
} // end constructor Increment

// print count and increment values
void Increment::print() const
{
    cout << "count = " << count << ", increment = " << increment << endl;
} // end function print
```

**Fig. 18.5** | Member initializer used to initialize a constant of a built-in data type. (Part 2 of 2.)

```cpp
// Fig. 18.6: fig18_06.cpp
// Program to test class Increment.
#include <iostream>
#include "Increment.h" // include definition of class Increment
using namespace std;

int main()
{
    Increment value( 10, 5 );
    cout << "Before incrementing: ";
    value.print();

    for ( int j = 1; j <= 3; j++ )
    {
        value.addIncrement();
        cout << "After increment " << j << " : ";
        value.print();
    } // end for
} // end main
```

**Fig. 18.6** | Invoking an Increment object’s `print` and `addIncrement` member functions.

The constructor definition (Fig. 18.5, lines 9–14) uses a **member initializer list** to initialize class `Increment`’s data members—non-const integer `count` and const integer `increment` (declared in lines 19–20 of Fig. 18.4). Member initializers appear between a constructor’s parameter list and the left brace that begins the constructor’s body. The
member initializer list (Fig. 18.5, lines 10–11) is separated from the parameter list with a colon (:) Each member initializer consists of the data member name followed by parentheses containing the member’s initial value. In this example, count is initialized with the value of constructor parameter c and increment is initialized with the value of constructor parameter i. Multiple member initializers are separated by commas. Also, the member initializer list executes before the body of the constructor executes.

Software Engineering Observation 18.3
A const object cannot be modified by assignment, so it must be initialized. When a data member of a class is declared const, a member initializer must be used to provide the constructor with the initial value of the data member for an object of the class. The same is true for references.

Erroneously Attempting to Initialize a const Data Member with an Assignment
The program of Figs. 18.7–18.9 illustrates the compilation errors caused by attempting to initialize const data member increment with an assignment statement (Fig. 18.8, line 12) in the Increment constructor’s body rather than with a member initializer. Line 11 of Fig. 18.8 does not generate a compilation error, because count is not declared const.

Common Programming Error 18.5
Not providing a member initializer for a const data member is a compilation error.

Software Engineering Observation 18.4
Constant data members (const objects and const variables) and data members declared as references must be initialized with member initializer syntax; assignments for these types of data in the constructor body are not allowed.

Fig. 18.7 Increment class definition containing non-const data member count and const data member increment. (Part 1 of 2.)


```cpp
const int increment; // const data member
}; // end class Increment
#endif
```

**Fig. 18.7** | Increment class definition containing non-const data member count and const data member increment. (Part 2 of 2.)

```cpp
// Fig. 18.8: Increment.cpp
// Erroneous attempt to initialize a constant of a built-in data type by assignment.
#include <iostream>
#include "Increment.h" // include definition of class Increment
using namespace std;

// constructor; constant member 'increment' is not initialized
Increment::Increment( int c, int i )
{
    count = c; // allowed because count is not constant
    increment = i; // ERROR: Cannot modify a const object
} // end constructor Increment

// print count and increment values
void Increment::print() const
{
    cout << "count = " << count << ", increment = " << increment << endl;
} // end function print
```

**Fig. 18.8** | Erroneous attempt to initialize a constant of a built-in data type by assignment.

```cpp
int main()
{
    Increment value( 10, 5 );
    cout << "Before incrementing: ";
    value.print();
    for ( int j = 1; j <= 3; j++ )
    {
        value.addIncrement();
        cout << "After increment " << j << " : ";
        value.print();
    } // end for
} // end main
```

**Fig. 18.9** | Program to test class Increment generates compilation errors. (Part 1 of 2.)
18.3 Composition: Objects as Members of Classes

Function print (Fig. 18.8, lines 16–19) is declared const. It might seem strange to label this function const, because a program probably will never have a const Increment object. However, it’s possible that a program will have a const reference to an Increment object or a pointer to const that points to an Increment object. Typically, this occurs when objects of class Increment are passed to functions or returned from functions. In these cases, only class Increment’s const member functions can be called through the reference or pointer. Thus, it’s reasonable to declare function print as const—doing so prevents errors in these situations where an Increment object is treated as a const object.

Error-Prevention Tip 18.1
Declare as const all of a class’s member functions that do not modify the object in which they operate. Occasionally this may seem inappropriate, because you’ll have no intention of creating const objects of that class or accessing objects of that class through const references or pointers to const. Declaring such member functions const does offer a benefit, though. If the member function is inadvertently written to modify the object, the compiler will issue an error message.

18.3 Composition: Objects as Members of Classes

An AlarmClock object needs to know when it’s supposed to sound its alarm, so why not include a Time object as a member of the AlarmClock class? Such a capability is called composition and is sometimes referred to as a has-a relationship—a class can have objects of other classes as members.

Software Engineering Observation 18.5
A common form of software reusability is composition, in which a class has objects of other classes as members.

When an object is created, its constructor is called automatically. Previously, we saw how to pass arguments to the constructor of an object we created in main. This section shows how an object’s constructor can pass arguments to member-object constructors via member initializers.
The next program uses classes Date (Figs. 18.10–18.11) and Employee (Figs. 18.12–18.13) to demonstrate composition. Class Employee’s definition (Fig. 18.12) contains private data members firstName, lastName, birthDate and hireDate. Members birthDate and hireDate are const objects of class Date, which contains private data members month, day and year. The Employee constructor’s header (Fig. 18.13, lines 10–11) specifies that the constructor has four parameters (first, last, dateOfBirth and dateOfHire). The first two parameters are passed via member initializers to the string class constructor. The last two are passed via member initializers to the Date class constructor.

### Software Engineering Observation 18.6

Member objects are constructed in the order in which they’re declared in the class definition (not in the order they’re listed in the constructor’s member initializer list) and before their enclosing class objects (sometimes called host objects) are constructed.

---

```cpp
1 // Fig. 18.10: Date.h
2 // Date class definition; Member functions defined in Date.cpp
3 #ifndef DATE_H
4 #define DATE_H
5
6 class Date
7 {
8  public:
9    static const int monthsPerYear = 12; // number of months in a year
10    Date( int = 1, int = 1, int = 1900 ); // default constructor
11    void print() const; // print date in month/day/year format
12  ~Date(); // provided to confirm destruction order
13  private:
14    int month; // 1-12 (January-December)
15    int day; // 1-31 based on month
16    int year; // any year
17    // utility function to check if day is proper for month and year
18    int checkDay( int ) const;
19  }; // end class Date
20 #endif
```

---

```cpp
1 // Fig. 18.11: Date.cpp
2 // Date class member-function definitions.
3 #include <iostream>
4 #include "Date.h" // include Date class definition
5 using namespace std;
6
7 // constructor confirms proper value for month; calls
8 // utility function checkDay to confirm proper value for day
```
Date::Date( int mn, int dy, int yr )
{
    if ( mn > 0 && mn <= monthsPerYear ) // validate the month
        month = mn;
    else // invalid month set to 1
    {
        month = 1;
        cout << "Invalid month (" << mn << ") set to 1.\n";
    } // end else

    year = yr; // could validate yr
    day = checkDay( dy ); // validate the day

    // output Date object to show when its constructor is called
    cout << "Date object constructor for date ";
    print();
    cout << endl;
} // end Date constructor

// print Date object in form month/day/year
void Date::print() const
{
    cout << month << '/' << day << '/' << year;
} // end function print

// output Date object to show when its destructor is called
Date::~Date()
{
    cout << "Date object destructor for date ";
    print();
    cout << endl;
} // end ~Date destructor

// utility function to confirm proper day value based on
// month and year; handles leap years, too
int Date::checkDay( int testDay ) const
{
    static const int daysPerMonth[ monthsPerYear + 1 ] =
    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }; 

    // determine whether testDay is valid for specified month
    if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
        return testDay;

    // February 29 check for leap year
    if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
        ( year % 4 == 0 && year % 100 != 0 ) ) )
        return testDay;

    cout << "Invalid day (" << testDay << ") set to 1.\n";
    return 1; // leave object in consistent state if bad value
} // end function checkDay
// Fig. 18.12: Employee.h
// Employee class definition showing composition.
// Member functions defined in Employee.cpp.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
#include "Date.h" // include Date class definition
using namespace std;

class Employee
{
public:
    Employee( const string &, const string &,
               const Date &, const Date & );
    void print() const;
    ~Employee(); // provided to confirm destruction order

private:
    string firstName; // composition: member object
    string lastName; // composition: member object
    const Date birthDate; // composition: member object
    const Date hireDate; // composition: member object
}; // end class Employee
#endif

Fig. 18.12 | Employee class definition showing composition.

// Fig. 18.13: Employee.cpp
// Employee class member-function definitions.
#include <iostream>
#include "Employee.h" // Employee class definition
#include "Date.h" // Date class definition
using namespace std;

// constructor uses member initializer list to pass initializer
// values to constructors of member objects
Employee::Employee( const string &first, const string &last,
                     const Date &dateOfBirth, const Date &dateOfHire )
    : firstName( first ), // initialize firstName
      lastName( last ), // initialize lastName
      birthDate( dateOfBirth ), // initialize birthDate
      hireDate( dateOfHire ) // initialize hireDate
{
    // output Employee object to show when constructor is called
    cout << "Employee object constructor: "
         << firstName << ' ' << lastName << endl;
} // end Employee constructor

Fig. 18.13 | Employee class member-function definitions, including constructor with a member initializer list. (Part 1 of 2.)
Employee Constructor's Member Initializer List

The colon (:) following the constructor’s header (Fig. 18.13, line 12) begins the member initializer list. The member initializers specify the Employee constructor parameters being passed to the constructors of the string and Date data members. Parameters firstName, lastName, dateOfBirth, and dateOfHire are passed to the constructors for objects firstName’s (Fig. 18.13, line 12), lastName (Fig. 18.13, line 13), birthDate (Fig. 18.13, line 14) and hireDate (Fig. 18.13, line 15), respectively. Again, member initializers are separated by commas.

Date Class’s Default Copy Constructor

As you study class Date (Fig. 18.10), notice that the class does not provide a constructor that receives a parameter of type Date. So, why can the Employee constructor’s member initializer list initialize the birthDate and hireDate objects by passing Date object’s to their Date constructors? As we mentioned in Chapter 17, the compiler provides each class with a default copy constructor that copies each data member of the constructor’s argument object into the corresponding member of the object being initialized. Chapter 19 discusses how you can define customized copy constructors.

Testing Classes Date and Employee

Figure 18.14 creates two Date objects (lines 9–10) and passes them as arguments to the constructor of the Employee object created in line 11. Line 14 outputs the Employee object’s data. When each Date object is created in lines 9–10, the Date constructor defined in lines 9–26 of Fig. 18.11 displays a line of output to show that the constructor was called (see the first two lines of the sample output). [Note: Line 11 of Fig. 18.14 causes two additional Date constructor calls that do not appear in the program’s output. When each of the Employee’s Date member object’s is initialized in the Employee constructor’s member initializer list (Fig. 18.13, lines 14–15), the default copy constructor for class Date is called. Since this constructor is defined implicitly by the compiler, it does not contain any output statements to demonstrate when it’s called.]
Class `Date` and class `Employee` each include a destructor (lines 35–40 of Fig. 18.11 and lines 33–37 of Fig. 18.13, respectively) that prints a message when an object of its class is destructed. This enables us to confirm in the program output that objects are constructed from the `inside out` and destroyed in the reverse order, from the `outside in` (i.e., the `Date` member objects are destroyed after the `Employee` object that contains them). Notice the last four lines in the output of Fig. 18.14. The last two lines are the outputs of the `Date` destructor running on `Date` objects `hire` (line 10) and `birth` (line 9), respectively. These outputs confirm that the three objects created in `main` are destructed in the `reverse` of the order in which they were constructed. The `Employee` destructor output is five lines from the bottom. The fourth and third lines from the bottom of the output window show the destructors running for the `Employee`'s member objects `hireDate` (Fig. 18.12, line 22) and `birthDate` (Fig. 18.12, line 21). These outputs confirm that the `Employee` object is destructed from the `outside in`—i.e., the `Employee` destructor runs first (output shown five
lines from the bottom of the output window), then the member objects are destructed in the reverse order from which they were constructed. Class string’s destructor does not contain output statements, so we do not see the firstName and lastName objects being destructed. Again, Fig. 18.14’s output did not show the constructors running for member objects birthDate and hireDate, because these objects were initialized with the default Date class copy constructors provided by the compiler.

What Happens When I Do Not Use the Member Initializer List?
If a member object is not initialized through a member initializer, the member object’s default constructor will be called implicitly. Values, if any, established by the default constructor can be overridden by set functions. However, for complex initialization, this approach may require significant additional work and time.

Common Programming Error 18.6
A compilation error occurs if a member object is not initialized with a member initializer and the member object’s class does not provide a default constructor (i.e., the member object’s class defines one or more constructors, but none is a default constructor).

Performance Tip 18.2
Initialize member objects explicitly through member initializers. This eliminates the overhead of “doubly initializing” member objects—once when the member object’s default constructor is called and again when set functions are called in the constructor body (or later) to initialize the member object.

Software Engineering Observation 18.7
If a class member is an object of another class, making that member object public does not violate the encapsulation and hiding of that member object’s private members. But, it does violate the encapsulation and hiding of the containing class’s implementation, so member objects of class types should still be private, like all other data members.

18.4 friend Functions and friend Classes

A friend function of a class is defined outside that class’s scope, yet has the right to access the non-public (and public) members of the class. Standalone functions, entire classes or member functions of other classes may be declared to be friends of another class.

Using friend functions can enhance performance. This section presents a mechanical example of how a friend function works. Later in the book, friend functions are used to overload operators for use with class objects (Chapter 19) and to create iterator classes. Objects of an iterator class can successively select items or perform an operation on items in a container class object. Objects of container classes can store items. Using friends is often appropriate when a member function cannot be used for certain operations, as we’ll see in Chapter 19.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend. To declare all member functions of class ClassTwo as friends of class ClassOne, place a declaration of the form

```cpp
friend class ClassTwo;
```

in the definition of class ClassOne.
Friendship is granted, not taken—i.e., for class B to be a friend of class A, class A must explicitly declare that class B is its friend. Also, the friendship relation is neither symmetric nor transitive; i.e., if class A is a friend of class B, and class B is a friend of class C, you cannot infer that class B is a friend of class A (again, friendship is not symmetric), that class C is a friend of class B (also because friendship is not symmetric), or that class A is a friend of class C (friendship is not transitive).

Modifying a Class’s private Data with a Friend Function
Figure 18.15 is a mechanical example in which we define friend function setX to set the private data member x of class Count. The friend declaration (line 9) appears first (by convention) in the class definition, even before public member functions are declared. Again, this friend declaration can appear anywhere in the class.

```
1  // Fig. 18.15: fig18_15.cpp
2  // Friends can access private members of a class.
3  #include <iostream>
4  using namespace std;
5  
6  // Count class definition
7  class Count
8  {
9      friend void setX( Count &, int ); // friend declaration
10  public:
11      // constructor
12      Count()
13      : x( 0 ) // initialize x to 0
14      {
15          // empty body
16      } // end constructor Count
17
```

Fig. 18.15 | Friends can access private members of a class. (Part 1 of 2.)
Function `setX` (lines 29–32) is a C-style, stand-alone function—it isn’t a member function of class `Count`. For this reason, when `setX` is invoked for object `counter`, line 41 passes `counter` as an argument to `setX` rather than using a handle (such as the name of the object) to call the function, as in

```cpp
setX( counter, 8 );
```

If you remove the friend declaration in line 9, you’ll receive error messages indicating that function `setX` cannot modify class `Count`’s private data member `x`.

As we mentioned, Fig. 18.15 is a mechanical example of using the friend construct. It would normally be appropriate to define function `setX` as a member function of class `Count`. It would also normally be appropriate to separate the program of Fig. 18.15 into three files:

1. A header file (e.g., `Count.h`) containing the `Count` class definition, which in turn contains the prototype of friend function `setX`
2. An implementation file (e.g., `Count.cpp`) containing the definitions of class `Count`’s member functions and the definition of friend function `setX`
3. A test program (e.g., `fig18_15.cpp`) with `main`.  

Fig. 18.15  |  Friends can access private members of a class. (Part 2 of 2.)
Overloaded friend Functions

It’s possible to specify overloaded functions as friends of a class. Each function intended to be a friend must be explicitly declared in the class definition as a friend of the class.

18.5 Using the this Pointer

We’ve seen that an object’s member functions can manipulate the object’s data. How do member functions know which object’s data members to manipulate? Every object has access to its own address through a pointer called this (a C++ keyword). The this pointer is not part of the object itself—i.e., the memory occupied by the this pointer is not reflected in the result of a sizeof operation on the object. Rather, the this pointer is passed (by the compiler) as an implicit argument to each of the object’s non-static member functions. Section 18.6 introduces static class members and explains why the this pointer is not implicitly passed to static member functions.

Objects use the this pointer implicitly (as we’ve done to this point) or explicitly to reference their data members and member functions. The type of the this pointer depends on the type of the object and whether the member function in which this is used is declared const. For example, in a nonconstant member function of class Employee, the this pointer has type Employee * const (a constant pointer to a nonconstant Employee object). In a constant member function of the class Employee, the this pointer has the data type const Employee * const (a constant pointer to a constant Employee object).

The next example shows implicit and explicit use of the this pointer; later in this chapter and in Chapter 19, we show some substantial and subtle examples of using this.

Implicitly and Explicitly Using the this Pointer to Access an Object’s Data Members

Figure 18.16 demonstrates the implicit and explicit use of the this pointer to enable a member function of class Test to print the private data x of a Test object.

```cpp
1 // Fig. 18.16: fig18_16.cpp
2 // Using the this pointer to refer to object members.
3 #include <iostream>
4 using namespace std;
5
6 class Test
7 {
8   public:
9     Test( int = 0 ); // default constructor
10     void print() const;
11   private:
12     int x;
13   }; // end class Test
14
15 // constructor
16 Test::Test( int value )
17   : x( value ) // initialize x to value
18 {
19     // empty body
20   } // end constructor Test
```

Fig. 18.16 | this pointer implicitly and explicitly accessing an object’s members. (Part 1 of 2.)
18.5 Using the `this` Pointer

For illustration purposes, member function `print` (lines 24–36) first prints `x` by using the `this` pointer implicitly (line 27)—only the name of the data member is specified. Then `print` uses two different notations to access `x` through the `this` pointer—the arrow operator (`->`) off the `this` pointer (line 31) and the dot operator (`.`) off the dereferenced `this` pointer (line 35). Note the parentheses around `*this` (line 35) when used with the dot member selection operator (`.`). The parentheses are required because the dot operator has higher precedence than the `*` operator. Without the parentheses, the expression `*this.x` would be evaluated as if it were parenthesized as `*(this.x)`, which is a compilation error, because the dot operator cannot be used with a pointer.

### Common Programming Error 18.7

Attempting to use the member selection operator (`.`) with a pointer to an object is a compilation error—the dot member selection operator may be used only with an lvalue such as an object’s name, a reference to an object or a dereferenced pointer to an object.

One interesting use of the `this` pointer is to prevent an object from being assigned to itself. As we’ll see in Chapter 19, self-assignment can cause serious errors when the object contains pointers to dynamically allocated storage.

### Using the `this` Pointer to Enable Cascaded Function Calls

Another use of the `this` pointer is to enable cascaded member-function calls—that is, invoking multiple functions in the same statement (as in line 12 of Fig. 18.19). The program
of Figs. 18.17–18.19 modifies class Time’s set functions setTime, setHour, setMinute and setSecond such that each returns a reference to a Time object to enable cascaded member-function calls. Notice in Fig. 18.18 that the last statement in the body of each of these member functions returns *this (lines 22, 29, 36 and 43) into a return type of Time &.

```cpp
1 // Fig. 18.17: Time.h
2 // Cascading member function calls.
3 // Time class definition.
4 // Member functions defined in Time.cpp.
5 #ifndef TIME_H
6 #define TIME_H
7 
8 class Time
9 {
10 public:
11     Time( int = 0, int = 0, int = 0 ); // default constructor
12     // set functions (the Time & return types enable cascading)
13     Time &setTime( int, int, int ); // set hour, minute, second
14     Time &setHour( int ); // set hour
15     Time &setMinute( int ); // set minute
16     Time &setSecond( int ); // set second
17     // get functions (normally declared const)
18     int getHour() const; // return hour
19     int getMinute() const; // return minute
20     int getSecond() const; // return second
21     // print functions (normally declared const)
22     void printUniversal() const; // print universal time
23     void printStandard() const; // print standard time
24 
25 private:
26     int hour; // 0 - 23 (24-hour clock format)
27     int minute; // 0 - 59
28     int second; // 0 - 59
29 }; // end class Time
30 
31#endif
```

**Fig. 18.17** | Time class definition modified to enable cascaded member-function calls.

```cpp
1 // Fig. 18.18: Time.cpp
2 // Time class member-function definitions.
3 #include <iostream>
4 #include <iomanip>
5 #include "Time.h" // Time class definition
6 using namespace std;
7
```

**Fig. 18.18** | Time class member-function definitions modified to enable cascaded member-function calls. (Part 1 of 3.)
// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hr, int min, int sec )
{
    setTime( hr, min, sec );
} // end Time constructor

// set values of hour, minute, and second
Time &Time::setTime( int h, int m, int s ) // note Time & return
{
    setHour( h );
    setMinute( m );
    setSecond( s );
    return *this; // enables cascading
} // end function setTime

// set hour value
Time &Time::setHour( int h ) // note Time & return
{
    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
    return *this; // enables cascading
} // end function setHour

// set minute value
Time &Time::setMinute( int m ) // note Time & return
{
    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
    return *this; // enables cascading
} // end function setMinute

// set second value
Time &Time::setSecond( int s ) // note Time & return
{
    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
    return *this; // enables cascading
} // end function setSecond

// get hour value
int Time::getHour() const
{
    return hour;
} // end function getHour

// get minute value
int Time::getMinute() const
{
    return minute;
} // end function getMinute

---

**Fig. 18.18**  |  Time class member-function definitions modified to enable cascaded member-function calls. (Part 2 of 3.)
// get second value
int Time::getSecond() const
{
    return second;
} // end function getSecond

// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal() const
{
    cout << setfill('0') << setw(2) << hour << ":
        " << setw(2) << minute << ":" << setw(2) << second;
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard() const
{
    cout << ( (hour == 0 || hour == 12) ? 12 : hour % 12 ) << ":" << setfill('0') << setw(2) << minute
        << ":" << setw(2) << second << (hour < 12 ? " AM" : " PM");
} // end function printStandard

// Fig. 18.18 | Time class member-function definitions modified to enable cascaded member-
function calls. (Part 3 of 3.)

// Fig. 18.19: fig18_19.cpp
// Cascading member-function calls with the this pointer.
#include <iostream>
#include "Time.h" // Time class definition
using namespace std;

int main()
{
    Time t; // create Time object

    // cascaded function calls
    t.setHour(18).setMinute(30).setSecond(22);
    cout << "Universal time: ";
    t.printUniversal();
    cout << "\nStandard time: ";
    t.printStandard();
    cout << "\n\nNew standard time: ";
    // cascaded function calls
    t.setTime(20, 20, 20).printStandard();
    cout << endl;
} // end main

// Fig. 18.19 | Cascading member-function calls with the this pointer. (Part 1 of 2.)
The program of Fig. 18.19 creates a `Time` object `t` (line 9), then uses it in cascaded member-function calls (lines 12 and 24). Why does the technique of returning `*this` as a reference work? The dot operator (.) associates from left to right, so line 12 first evaluates `t.setHour(18)`, then returns a reference to object `t` as the value of this function call. The remaining expression is then interpreted as

\[ t.setMinute(30).setSecond(22); \]

The `t.setMinute(30)` call executes and returns a reference to the object `t`. The remaining expression is interpreted as

\[ t.setSecond(22); \]

Line 24 also uses cascading. The calls must appear in the order shown in line 24, because `printStandard` as defined in the class does not return a reference to `t`. Placing the call to `printStandard` before the call to `setTime` in line 24 results in a compilation error. Chapter 19 presents several practical examples of using cascaded function calls. One such example uses multiple `<<` operators with `cout` to output multiple values in a single statement.

### 18.6 static Class Members

There is an important exception to the rule that each object of a class has its own copy of all the data members of the class. In certain cases, only one copy of a variable should be shared by all objects of a class. A `static` data member is used for these and other reasons. Such a variable represents “class-wide” information (i.e., a property that is shared by all instances and is not specific to any one object of the class).

#### Motivating Class-Wide Data

Let’s further motivate the need for static class-wide data with an example. Suppose that we have a video game with Martians and other space creatures. Each Martian tends to be brave and willing to attack other space creatures when the Martian is aware that there are at least five Martians present. If fewer than five are present, each Martian becomes cowardly. So each Martian needs to know the `martianCount`. We could endow each instance of class `Martian` with `martianCount` as a data member. If we do, every Martian will have a separate copy of the data member. Every time we create a new Martian, we’ll have to update the data member `martianCount` in all Martian objects. Doing this would require every Martian object to have, or have access to, handles to all other Martian objects in memory. This wastes space with the redundant copies and wastes time in updating the separate copies. Instead, we declare `martianCount` to be `static`. This makes `martianCount` class-wide data. Every Martian can access `martianCount` as if it were a data member of the Martian, but only one copy of the static variable `martianCount` is maintained by C++.
This saves space. We save time by having the Martian constructor increment static variable martianCount and having the Martian destructor decrement martianCount. Because there is only one copy, we do not have to increment or decrement separate copies of martianCount for each Martian object.

**Performance Tip 18.3**

Use static data members to save storage when a single copy of the data for all objects of a class will suffice.

**Scope and Initialization of static Data Members**

Although they may seem like global variables, a class’s static data members have class scope. Also, static members can be declared public, private or protected. A fundamental-type static data member is initialized by default to 0. If you want a different initial value, a static data member can be initialized once. A static const data member of int or enum type can be initialized in its declaration in the class definition. However, all other static data members must be defined at global namespace scope (i.e., outside the body of the class definition) and can be initialized only in those definitions. If a static data member is an object of a class that provides a default constructor, the static data member need not be initialized because its default constructor will be called.

**Accessing static Data Members**

A class’s private and protected static members are normally accessed through the class’s public member functions or friends. A class’s static members exist even when no objects of that class exist. To access a public static class member when no objects of the class exist, simply prefix the class name and the binary scope resolution operator (::) to the name of the data member. For example, if our preceding variable martianCount is public, it can be accessed with the expression Martian::martianCount when there are no Martian objects. (Of course, using public data is discouraged.)

To access a private or protected static class member when no objects of the class exist, provide a public static member function and call the function by prefixing its name with the class name and binary scope resolution operator. A static member function is a service of the class, not of a specific object of the class.

**Software Engineering Observation 18.11**

A class’s static data members and static member functions exist and can be used even if no objects of that class have been instantiated.

**Demonstrating static Data Members**

The program of Figs. 18.20–18.22 demonstrates a private static data member called count (Fig. 18.20, line 25) and a public static member function called getCount (Fig. 18.20, line 19). In Fig. 18.21, line 8 defines and initializes the data member count to zero at global namespace scope and lines 12–15 define static member function getCount. Notice that neither line 8 nor line 12 includes keyword static, yet both lines refer to static class members. When static is applied to an item at global namespace scope, that item becomes known only in that file. The static class members need to be available to any client code that uses the class, so we declare them static only in the .h file. Data
member count maintains a count of the number of objects of class Employee that have been instantiated. When objects of class Employee exist, member count can be referenced through any member function of an Employee object—in Fig. 18.21, count is referenced by both line 22 in the constructor and line 32 in the destructor.

Common Programming Error 18.8

It's a compilation error to include keyword static in the definition of a static data member at global namespace scope.

```
1 // Fig. 18.20: Employee.h
2 // Employee class definition with a static data member to
3 // track the number of Employee objects in memory
4 #ifndef EMPLOYEE_H
5 #define EMPLOYEE_H
6
7 #include <string>
8 using namespace std;
9
10 class Employee
11 {
12 public:
13     Employee( const string &, const string & ); // constructor
14     ~Employee(); // destructor
15     string getFirstName() const; // return first name
16     string getLastName() const; // return last name
17
18     // static member function
19     static int getCount(); // return number of objects instantiated
20 private:
21     string firstName;
22     string lastName;
23
24     // static data
25     static int count; // number of objects instantiated
26 }; // end class Employee
27
#endif
```

Fig. 18.20  Employee class definition with a static data member to track the number of Employee objects in memory.

```
1 // Fig. 18.21: Employee.cpp
2 // Employee class member-function definitions.
3 #include <iostream>
4 #include "Employee.h" // Employee class definition
5 using namespace std;
6
```

Fig. 18.21  Employee class member-function definitions. (Part 1 of 2.)
Figure 18.22 uses static member function getCount to determine the number of Employee objects in memory at various points in the program. The program calls Employee::getCount() before any Employee objects have been created (line 12), after two Employee objects have been created (line 23) and after those Employee objects have been destroyed (line 34). Lines 16–29 in main define a nested scope. Recall that local variables exist until the scope in which they are defined terminates. In this example, we create two Employee objects in lines 17–18 inside the nested scope. As each constructor executes, it increments class Employee’s static data member count. These Employee objects are destroyed when the program reaches line 29. At that point, each object’s destructor executes and decrements class Employee’s static data member count.
A member function should be declared static if it does not access non-static data members or non-static member functions of the class. Unlike non-static member functions, a static member function does not have a this pointer, because static data mem-

```cpp
// Fig. 18.22: fig18_22.cpp
// static data member tracking the number of objects of a class.
#include <iostream>
#include "Employee.h"  // Employee class definition
using namespace std;

int main()
{
    // no objects exist; use class name and binary scope resolution
    // operator to access static member function getCount
    cout << "Number of employees before instantiation of any objects is "
        << Employee::getCount() << endl; // use class name

    // the following scope creates and destroys
    // Employee objects before main terminates
    {
        Employee e1( "Susan", "Baker" );
        Employee e2( "Robert", "Jones" );

        // two objects exist; call static member function getCount again
        // using the class name and the binary scope resolution operator
        cout << "Number of employees after objects are instantiated is "
            << Employee::getCount();

        cout << "\n\nEmployee 1: "
            << e1.getFirstName() << " " << e1.getLastName()
            << "\nEmployee 2: "
            << e2.getFirstName() << " " << e2.getLastName() << "\n\n";
    } // end nested scope in main

    // no objects exist, so call static member function getCount again
    // using the class name and the binary scope resolution operator
    cout << "Number of employees after objects are deleted is "
        << Employee::getCount() << endl;
} // end main
```

Number of employees before instantiation of any objects is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after objects are instantiated is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Robert Jones
~Employee() called for Susan Baker

Number of employees after objects are deleted is 0

Fig. 18.22 | static data member tracking the number of objects of a class.
bers and *static* member functions exist independently of any objects of a class. The *this* pointer must refer to a specific object of the class, and when a *static* member function is called, there might not be any objects of its class in memory.

**Common Programming Error 18.9**
*) Using the *this* pointer in a *static* member function is a compilation error.

**Common Programming Error 18.10**
*) Declaring a *static* member function *const* is a compilation error. The *const* qualifier indicates that a function cannot modify the contents of the object in which it operates, but *static* member functions exist and operate independently of any objects of the class.

### 18.7 Data Abstraction and Information Hiding

Classes normally hide the details of their implementation from their clients. This is called *information hiding*. As an example, let's consider the stack data structure. Recall that the stack is a last-in-first-out (LIFO) data structure—the last item pushed (inserted) on the stack is the first item popped (removed) off the stack.

Stacks can be implemented with arrays and with other data structures, such as linked lists. (We discuss stacks in Chapter 22.) A client of a stack class need not be concerned with the stack’s implementation. The client knows only that when data items are placed in the stack, they will be recalled in last-in, first-out order. The client cares about *what* functionality a stack offers, not about *how* that functionality is implemented. This concept is referred to as *data abstraction*. Although you might know the details of a class’s implementation, you should not write code that depends on these details as the details may later change. This enables a particular class (such as one that implements a stack and its operations, *push* and *pop*) to be replaced with another version without affecting the rest of the system. As long as the *public* services of the class do not change (i.e., every original *public* member function still has the same prototype in the new class definition), the rest of the system is not affected.

**Abstract Data Types**

Many programming languages emphasize actions. In these languages, data exists to support the actions that programs must take. Data is “less interesting” than actions. Data is “crude.” Only a few built-in data types exist, and it’s difficult to create new types. C++ and the object-oriented style of programming elevate the importance of data. The primary activities of object-oriented programming in C++ are the creation of types (i.e., classes) and the expression of the interactions among objects of those types. To create languages that emphasize data, the programming-languages community needed to formalize some notions about data. The formalization we consider here is the notion of *abstract data types* (ADTs), which improve the application-development process.

What’s an abstract data type? Consider the type `int`, which most people would associate with an integer in mathematics. Rather, an `int` is an abstract representation of an integer. Unlike mathematical integers, computer `ints` have a maximum size—one 32-bit machines is typically limited to the range from $-2,147,483,648$ to $+2,147,483,647$. If the result of a calculation falls outside this range, an “overflow” error occurs and the computer...
responds in some machine-dependent manner. It might, for example, “quietly” produce an incorrect result, such as a value too large to fit in an `int` variable (commonly called arithmetic overflow). Mathematical integers do not have this problem. Therefore, the notion of a computer `int` is only an approximation of the notion of a real-world integer.

Types like `int`, `double`, `char` and others are all examples of abstract data types. They’re essentially ways of representing real-world notions to some satisfactory level of precision within a computer system.

An abstract data type actually captures two notions—A data representation and the operations that can be performed on that data. For example, in C++, an `int` contains an integer value (data) and provides addition, subtraction, multiplication, division and modulus operations (among others)—division by zero is undefined. These allowed operations perform in a manner sensitive to machine parameters, such as the fixed word size of the underlying computer system. Another example is the notion of negative integers, whose operations and data representation are clear, but the operation of taking the square root of a negative integer is undefined. In C++, you can use classes to implement abstract data types and their services. For example, to implement a stack ADT, we create our own stack class in Chapter 22.

### Queue Abstract Data Type

Each of us stands in line from time to time. A waiting line is also called a queue. Computer systems use waiting lines internally, so we need to write programs that implement queues. A queue is another example of an abstract data type.

Queues offer well-understood behavior to their clients. Clients put things in a queue one at a time—by invoking the queue’s `enqueue` operation—and the clients get those things back one at a time on demand—by invoking the queue’s `dequeue` operation. Conceptually, a queue can become infinitely long; a real queue, of course, is finite. Items are returned from a queue in first-in, first-out (FIFO) order—the first item inserted in the queue is the first item removed from the queue.

The queue hides an internal data representation that keeps track of the items currently waiting in line, and offers a set of operations to clients, namely, `enqueue` and `dequeue`. The clients are not concerned about the implementation of the queue. Clients merely want the queue to operate “as advertised.” When a client enqueues a new item, the queue should accept that item and place it internally in some kind of first-in, first-out data structure. When the client wants the next item from the front of the queue, the queue should remove the item from its internal representation and deliver it to the client in FIFO order (i.e., the item that has been in the queue the longest should be the next one returned by the next `dequeue` operation).

The queue ADT guarantees the integrity of its internal data structure. Clients may not manipulate this data structure directly. Only the queue member functions have access to its internal data. Clients may cause only allowable operations to be performed on the data representation; operations not provided in the ADT’s public interface are rejected in some
appropriate manner. This could mean issuing an error message, throwing an exception (see Chapter 24), terminating execution or simply ignoring the operation request.

### 18.8 Wrap-Up

This chapter introduced several advanced topics related to classes and data abstraction. You learned how to specify `const` objects and `const` member functions to prevent modifications to objects, thus enforcing the principle of least privilege. You also learned that, through composition, a class can have objects of other classes as members. We introduced the topic of friendship and presented examples that demonstrate how to use `friend` functions.

You learned that the `this` pointer is passed as an implicit argument to each of a class’s non-static member functions, allowing the functions to access the correct object’s data members and other non-static member functions. You also saw explicit use of the `this` pointer to access the class’s members and to enable cascaded member-function calls. We motivated the need for static data members and demonstrated how to declare and use static data members and static member functions in your own classes.

You learned about data abstraction and information hiding—two of the fundamental concepts of object-oriented programming. Finally, we discussed abstract data types—ways of representing real-world or conceptual notions to some satisfactory level of precision within a computer system.

In Chapter 19, we continue our study of classes and objects by showing how to enable C++’s operators to work with objects—a process called operator overloading. For example, you’ll see how to “overload” the `<<` operator so it can be used to output a complete array without explicitly using a repetition statement.

---

**Summary**

**Section 18.2 const (Constant) Objects and const Member Functions**
- The keyword `const` can be used to specify that an object is not modifiable and that any attempt to modify the object should result in a compilation error.
- C++ compilers disallow non-`const` member function calls on `const` objects.
- An attempt by a `const` member function to modify an object of its class is a compilation error.
- A member function is specified as `const` both in its prototype and in its definition.
- A `const` object must be initialized.
- Constructors and destructors cannot be declared `const`.
- `const` data member and reference data members must be initialized using member initializers.

**Section 18.3 Composition: Objects as Members of Classes**
- A class can have objects of other classes as members—this concept is called composition.
- Member objects are constructed in the order in which they’re declared in the class definition and before their enclosing class objects are constructed.
- If a member initializer is not provided for a member object, the member object’s default constructor will be called implicitly.
Section 18.4 friend Functions and friend Classes

- A friend function of a class is defined outside that class’s scope, yet has the right to access all of the class’s members. Stand-alone functions or entire classes may be declared to be friends.
- A friend declaration can appear anywhere in the class.
- The friendship relation is neither symmetric nor transitive.

Section 18.5 Using the this Pointer

- Every object has access to its own address through the this pointer.
- An object’s this pointer is not part of the object itself—i.e., the size of the memory occupied by the this pointer is not reflected in the result of a sizeof operation on the object.
- The this pointer is passed as an implicit argument to each non-static member function.
- Objects use the this pointer implicitly (as we’ve done to this point) or explicitly to reference their data members and member functions.
- The this pointer enables cascaded member-function calls in which multiple functions are invoked in the same statement.

Section 18.6 static Class Members

- A static data member represents “class-wide” information (i.e., a property of the class shared by all instances, not a property of a specific object of the class).
- static data members have class scope and can be declared public, private or protected.
- A class’s static members exist even when no objects of that class exist.
- To access a public static class member when no objects of the class exist, simply prefix the class name and the binary scope resolution operator (::) to the name of the data member.
- A member function should be declared static if it does not access non-static data members or non-static member functions of the class. Unlike non-static member functions, a static member function does not have a this pointer, because static data members and static member functions exist independently of any objects of a class.

Section 18.7 Data Abstraction and Information Hiding

- Abstract data types are ways of representing real-world and conceptual notions to some satisfactory level of precision within a computer system.
- An abstract data type captures two notions: a data representation and the operations that can be performed on that data.

Terminology

- abstract data type (ADT) 664
- arithmetic overflow 665
- cascaded member-function calls 655
- composition 645
- data abstraction 664
- data representation 665
- dequeue (queue operation) 665
- enqueue (queue operation) 665
- first-in, first-out (FIFO) 665
- friend function 651
- has-a relationship 645
- host object 646
- information hiding 664
- member initializer 641
- member initializer list 642
- member initializer syntax 641
- member object 645
- member object constructor 645
- operations in an ADT 665
- queue 665
- queue abstract data type 665
- static data member 659
- static member function 660
- this pointer 654
Self-Review Exercises

18.1 Fill in the blanks in each of the following:
a) ______ must be used to initialize constant members of a class.
b) A nonmember function must be declared as a(n) ______ of a class to have access to that class’s private data members.
c) A constant object must be ______; it cannot be modified after it’s created.
d) A(n) ______ data member represents class-wide information.
e) An object’s non-static member functions have access to a “self pointer” to the object called the ______ pointer.
f) Keyword ______ specifies that an object or variable is not modifiable.
g) If a member initializer is not provided for a member object of a class, the object’s ______ is called.
h) A member function should be static if it does not access ______ class members.
i) Member objects are constructed ______ their enclosing class object.

18.2 Find the errors in the following class and explain how to correct them:

class Example
{
  public:
  Example( int y = 10 )
    : data( y )
  {
    // empty body
  } // end Example constructor

  int getIncrementedData() const
  {
    return data++;
  } // end function getIncrementedData

  static int getCount()
  {
    cout << "Data is " << data << endl;
    return count;
  } // end function getCount

private:
  int data;
  static int count;
}; // end class Example

Answers to Self-Review Exercises

18.1 a) member initializers. b) friend. c) initialized. d) static. e) this. f) const. g) default constructor. h) non-static. i) before.

18.2 Error: The class definition for Example has two errors. The first occurs in function getIncrementedData. The function is declared const, but it modifies the object.
Correction: To correct the first error, remove the const keyword from the definition of getIncrementedData.
Error: The second error occurs in function getCount. This function is declared static, so it isn’t allowed to access any non-static member (i.e., data) of the class.
Correction: To correct the second error, remove the output line from the getCount definition.

Exercises

18.3 Explain the notion of friendship. Explain the negative aspects of friendship as described in the text.
18.4 Can a correct Time class definition include both of the following constructors? If not, explain why not.

```cpp
Time( int h = 0, int m = 0, int s = 0 );
Time();
```

18.5 What happens when a return type, even void, is specified for a constructor or destructor?

18.6 (Date Class Modification) Modify class Date in Fig. 18.10 to have the following capabilities:

a) Output the date in multiple formats such as

- DDD YYYY
- MM/DD/YY
- June 14, 1992

b) Use overloaded constructors to create Date objects initialized with dates of the formats in part (a).

c) Create a Date constructor that reads the system date using the standard library functions of the <ctime> header and sets the Date members. (See your compiler's reference documentation or www.cplusplus.com/ref/ctime/index.html for information on the functions in header <ctime>.)

In Chapter 19, we'll be able to create operators for testing the equality of two dates and for comparing dates to determine whether one date is prior to, or after, another.

18.7 (SavingsAccount Class) Create a SavingsAccount class. Use a static data member annualInterestRate to store the annual interest rate for each of the savers. Each member of the class contains a private data member savingsBalance indicating the amount the saver currently has on deposit. Provide member function calculateMonthlyInterest that calculates the monthly interest by multiplying the balance by annualInterestRate divided by 12; this interest should be added to savingsBalance. Provide a static member function modifyInterestRate that sets the static annualInterestRate to a new value. Write a driver program to test class SavingsAccount. Instantiate two different objects of class SavingsAccount, saver1 and saver2, with balances of $2000.00 and $3000.00, respectively. Set the annualInterestRate to 3 percent. Then calculate the monthly interest and print the new balances for each of the savers. Then set the annualInterestRate to 4 percent, calculate the next month's interest and print the new balances for each of the savers.

18.8 (IntegerSet Class) Create class IntegerSet for which each object can hold integers in the range 0 through 100. Represent the set internally as a vector of bool values. Element a[i] is true if integer i is in the set. Element a[j] is false if integer j is not in the set. The default constructor initializes a set to the so-called “empty set,” i.e., a set for which all elements contain false.

Provide member functions for the common set operations. For example, provide a unionOfSets member function that creates a third set that is the set-theoretic union of two existing sets (i.e., an element of the result is set to true if that element is true in either or both of the existing sets, and an element of the result is set to false if that element is false in each of the existing sets).

Provide an intersectionOfSets member function which creates a third set which is the set-theoretic intersection of two existing sets (i.e., an element of the result is set to false if that element is false in either or both of the existing sets, and an element of the result is set to true if that element is true in each of the existing sets).

Provide an insertElement member function that places a new integer k into a set by setting a[k] to true. Provide a deleteElement member function that deletes integer m by setting a[m] to false.

Provide a printSet member function that prints a set as a list of numbers separated by spaces. Print only those elements that are present in the set (i.e., their position in the vector has a value of true). Print --- for an empty set.

Provide an isEqualTo member function that determines whether two sets are equal.

Provide an additional constructor that receives an array of integers and the size of that array and uses the array to initialize a set object.
Now write a driver program to test your IntegerSet class. Instantiate several IntegerSet objects. Test that all your member functions work properly.

18.9 (Time Class Modification) It would be perfectly reasonable for the Time class of Figs. 18.17–18.18 to represent the time internally as the number of seconds since midnight rather than the three integer values hour, minute and second. Clients could use the same public methods and get the same results. Modify the Time class of Fig. 18.17 to implement the time as the number of seconds since midnight and show that there is no visible change in functionality to the clients of the class. [Note: This exercise nicely demonstrates the virtues of implementation hiding.]

18.10 (Card Shuffling and Dealing) Create a program to shuffle and deal a deck of cards. The program should consist of class Card, class DeckOfCards and a driver program. Class Card should provide:

a) Data members face and suit of type int.
b) A constructor that receives two ints representing the face and suit and uses them to initialize the data members.
c) Two static arrays of strings representing the faces and suits.
d) A toString function that returns the Card as a string in the form “face of suit.” You can use the + operator to concatenate strings.

Class DeckOfCards should contain:

a) A vector of Cards named deck to store the Cards.
b) An integer currentCard representing the next card to deal.
c) A default constructor that initializes the Cards in the deck. The constructor should use vector function push_back to add each Card to the end of the vector after the Card is created and initialized. This should be done for each of the 52 Cards in the deck.
d) A shuffle function that shuffles the Cards in the deck. The shuffle algorithm should iterate through the vector of Cards. For each Card, randomly select another Card in the deck and swap the two Cards.
e) A dealCard function that returns the next Card object from the deck.
f) A moreCards function that returns a bool value indicating whether there are more Cards to deal.

The driver program should create a DeckOfCards object, shuffle the cards, then deal the 52 cards.

18.11 (Card Shuffling and Dealing) Modify the program you developed in Exercise 18.10 so that it deals a five-card poker hand. Then write functions to accomplish each of the following:

a) Determine whether the hand contains a pair.
b) Determine whether the hand contains two pairs.
c) Determine whether the hand contains three of a kind (e.g., three jacks).
d) Determine whether the hand contains four of a kind (e.g., four aces).
e) Determine whether the hand contains a flush (i.e., all five cards of the same suit).
f) Determine whether the hand contains a straight (i.e., five cards of consecutive face values).

Card Shuffling and Dealing Projects

18.12 (Card Shuffling and Dealing) Use the functions from Exercise 18.11 to write a program that deals two five-card poker hands, evaluates each hand and determines which is the better hand.

18.13 (Card Shuffling and Dealing) Modify the program you developed in Exercise 18.12 so that it can simulate the dealer. The dealer’s five-card hand is dealt “face down” so the player cannot see it. The program should then evaluate the dealer’s hand, and, based on the quality of the hand, the dealer should draw one, two or three more cards to replace the corresponding number of unneeded cards in the original hand. The program should then reevaluate the dealer’s hand.

18.14 (Card Shuffling and Dealing) Modify the program you developed in Exercise 18.13 so that it handles the dealer’s hand, but the player is allowed to decide which cards of the player’s hand to
replace. The program should then evaluate both hands and determine who wins. Now use this new program to play 20 games against the computer. Who wins more games, you or the computer? Have one of your friends play 20 games against the computer. Who wins more games? Based on the results of these games, make appropriate modifications to refine your poker-playing program. Play 20 more games. Does your modified program play a better game?

Making a Difference

18.15 (Air Traffic Control Project) Every day, according to the National Air Traffic Controllers Association (www.natca.org/mediacenter/bythenumbers.msp), there are more than 87,000 flights in the United States, including commercial flights, cargo flights, and so on, and the long-term trend is that air traffic activity will increase along with the population. As air traffic grows, so do the challenges to air traffic controllers, who monitor the flights and provide instructions to the pilots to ensure safety in the skies.

In this exercise, you’ll create a Flight class that could be used in a simple air-traffic-control simulator. The application’s main function will act as air traffic control. Visit sites such as www.howstuffworks.com/air-traffic-control.htm to research how the air-traffic-control system works. Then identify some key attributes of a Flight in an air-traffic-control system. Think about the different states a plane could be in from the time it’s parked at an airport gate until it arrives at its destination—parked, taxiing, waiting to take off, taking off, climbing, and so on. Use a FlightStatus enumeration to represent these states. The attributes might include the plane’s make and model, current air speed, current altitude, direction, carrier, departure time, estimated arrival time, origin and destination. The origin and destination should be specified using standard three-letter airport codes, such as BOS for Boston and LAX for Los Angeles (these codes are available at world-airport-codes.com). Provide set and get functions to manipulate these and any other attributes you identify. Next, identify the class’s behaviors and implement them as functions of the class. Include behaviors such as changeAltitude, reduceSpeed and beginLandingApproach. The Flight constructor should initialize a Flight’s attributes. You should also provide a toString function that returns a string representation of a Flight’s current status (e.g., parked at the gate, taxiing, taking off, changing altitude). This string should include all of the object’s instance-variable values.

When the application executes, main will display the message, "Air Traffic Control Simulator", then will create and interact with three Flight objects representing planes that are currently flying or preparing to fly. For simplicity, the Flight’s confirmation of each action will be a message displayed on the screen when the appropriate function is called on the object. For example, if you call a flight’s changeAltitude function, the method should:

a) Display a message containing the airline, flight number, "changing altitude", the current altitude and the new altitude.

b) Change the state of the status data member to CHANGING_ALTITUDE.

c) Change the value of the newAltitude data member.

In main, create and initialize three Flight objects that are in different states—for example, one could be at the gate, one could be preparing for takeoff and one could be preparing for landing. The main function should send messages to (invoke functions on) the Flight objects. As a Flight object receives each message, it should display a confirmation message from the function being called—such as “[Airline name] [Flight number] changing altitude from 20000 to 25000 feet.” The function should also update the appropriate state information in the Flight object. For example, if Air Traffic Control sends a message like “[Airline] [flight number] descend to 12000 feet,” the program should execute a function call like flight1.changeAltitude(12000), which would display a confirmation message and would set data member newAltitude to 12000. [Note: Assume the Flight’s currentAltitude data member is being set automatically by the plane’s altimeter.]
The whole difference between construction and creation is exactly this: that a thing constructed can only be loved after it is constructed; but a thing created is loved before it exists.
—Gilbert Keith Chesterton

Our doctor would never really operate unless it was necessary. He was just that way. If he didn’t need the money, he wouldn’t lay a hand on you.
—Herb Shriner

Objectives
In this chapter you’ll learn:

■ What operator overloading is and how it simplifies programming.
■ To overload operators for user-defined classes.
■ To overload unary and binary operators.
■ To convert objects from one class to another class.
■ To create PhoneNumber, Array and Date classes that demonstrate operator overloading.
■ To use overloaded operators and other features of C++’s string class.
■ To use keyword explicit to prevent the compiler from using single-argument constructors to perform implicit conversions.
19.1 Introduction

Chapters 17–18 introduced the basics of C++ classes. Services were obtained from objects by sending messages (in the form of member-function calls) to the objects. This function call notation is cumbersome for certain kinds of classes (such as mathematical classes). Also, many common manipulations are performed with operators (e.g., input and output). We can use C++’s rich set of built-in operators to specify common object manipulations. This chapter shows how to enable C++’s operators to work with objects—a process called operator overloading.

One example of an overloaded operator built into C++ is <<, which is used both as the stream insertion operator and as the bitwise left-shift operator (which is discussed in Chapter 10). Similarly, >> is also overloaded; it’s used both as the stream extraction operator and as the bitwise right-shift operator. Both of these operators are overloaded in the C++ Standard Library.

Although operator overloading sounds like an exotic capability, most programmers implicitly use overloaded operators. For example, the C++ language overloads the addition operator (+) and the subtraction operator (−). These operators perform differently, depending on their context in integer, floating-point and pointer arithmetic.

C++ enables you to overload most operators to be sensitive to the context in which they’re used—the compiler generates the appropriate code based on the context (in particular, the types of the operands). Some operators are overloaded frequently, especially the assignment, relational and various arithmetic operators such as + and −. The jobs performed by overloaded operators can also be performed by explicit function calls, but operator notation is often clearer and more familiar to programmers.

We discuss when to, and when not to, use operator overloading. We create classes PhoneNumber, Array and Date to demonstrate how to overload operators, including the stream insertion, stream extraction, assignment, equality, relational, subscript, logical negation and increment operators. We demonstrate C++’s Standard Library class string, which provides many overloaded operators. In the exercises, we ask you to implement several classes with overloaded operators. The exercises also use classes Complex (for complex numbers) and HugeInt (for integers larger than a computer can represent with type long) to demonstrate overloaded arithmetic operators + and −, and ask you to enhance those
classes by overloading other arithmetic operators. Finally, we show how to create a proxy class to hide a class’s implementation details (including its private data) from its clients.

19.2 Fundamentals of Operator Overloading

C++ programming is a type-sensitive and type-focused process. You can use fundamental types and can define new types. The fundamental types can be used with C++’s rich collection of operators. Operators provide you with a concise notation for expressing manipulations of data of fundamental types.

You can use operators with user-defined types as well. Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they’re used with objects, they have meaning appropriate to those objects.

An operator is overloaded by writing a non-static member function definition or global function definition as you normally would, except that the function name now becomes the keyword `operator` followed by the symbol for the operator being overloaded. For example, the function name `operator+` would be used to overload the addition operator (+). When operators are overloaded as member functions, they must be non-static, because they must be called on an object of the class and operate on that object.

To use an operator on class objects, that operator must be overloaded—with three exceptions. The assignment operator (=) may be used with every class to perform memberwise assignment of the class’s data members—each data member is assigned from the assignment’s “source” object to the “target” object. Memberwise assignment is dangerous for classes with pointer members; we’ll explicitly overload the assignment operator for such classes. The address (&) and comma (,) operators may also be used with objects of any class without overloading. The address operator returns a pointer to the object. The comma operator evaluates the expression to its left then the expression to its right, and returns the value of the latter expression. Both of these operators can also be overloaded.

Overloading is especially appropriate for mathematical classes. These often require that a substantial set of operators be overloaded to ensure consistency with the way these mathematical classes are handled in the real world. For example, it would be unusual to overload only addition for a complex number class, because other arithmetic operators are also commonly used with complex numbers.
19.3 Restrictions on Operator Overloading

Operator overloading provides the same concise and familiar expressions for user-defined types that C++ provides with its rich collection of operators for fundamental types. Operator overloading is not automatic—you must write operator-overloading functions to perform the desired operations. Sometimes these functions are best made member functions; sometimes they’re best as friend functions; occasionally they can be made global, non-friend functions. We present examples of each of these possibilities.

19.3 Restrictions on Operator Overloading

Most of C++’s operators can be overloaded. These are shown in Fig. 19.1. Figure 19.2 shows the operators that cannot be overloaded.

### Operators that can be overloaded

| + | - | * | / | % | ^ | & | | |
| ~ | ! | = | < | > | += | -= | *= |
| /= | %= | ^= | &= | |= | <<= | >>= | >>= |
| <<= | <= | => | >= | && | || | ++ |
| -- | ->* | , | -> | [ ] | () | new | delete |

Fig. 19.1 | Operators that can be overloaded.

### Operators that cannot be overloaded

| . | .* | :: | :? |

Fig. 19.2 | Operators that cannot be overloaded.

**Precedence, Associativity and Number of Operands**

The precedence of an operator cannot be changed by overloading. This can lead to awkward situations in which an operator is overloaded in a manner for which its fixed precedence is inappropriate. However, parentheses can be used to force the order of evaluation of overloaded operators in an expression.

The associativity of an operator (i.e., whether the operator is applied right-to-left or left-to-right) cannot be changed by overloading.

It isn’t possible to change the “arity” of an operator (i.e., the number of operands an operator takes): Overloaded unary operators remain unary operators; overloaded binary operators remain binary operators. C++’s only ternary operator (?:) cannot be overloaded. Operators &, *, + and - all have both unary and binary versions; these unary and binary versions can each be overloaded.

**Common Programming Error 19.1**

Attempting to change the “arity” of an operator via operator overloading is a compilation error.
Creating New Operators
It isn’t possible to create new operators; only existing operators can be overloaded. Unfortunately, this prevents you from using popular notations like the ** operator used in some other programming languages for exponentiation. [Note: You could overload an existing operator to perform exponentiation.]

Operators for Fundamental Types
The meaning of how an operator works on objects of fundamental types cannot be changed by operator overloading. You cannot, for example, change the meaning of how + adds two integers. Operator overloading works only with objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type.

Related Operators
Overloading an assignment operator and an addition operator to allow statements like

\[
\text{object2} = \text{object2} + \text{object1};
\]

does not imply that the += operator is also overloaded to allow statements such as

\[
\text{object2} += \text{object1};
\]

Such behavior can be achieved only by explicitly overloading operator += for that class.

19.4 Operator Functions as Class Members vs. Global Function
Operator functions can be member functions or global functions; global functions are often made friends for performance reasons. Member functions use the this pointer implicitly to obtain one of their class object arguments (the left operand for binary operators). Arguments for both operands of a binary operator must be explicitly listed in a global function call.
Operators That Must Be Overloaded as Member Functions

When overloading (), [], -> or any of the assignment operators, the operator overloading function must be declared as a class member. For the other operators, the operator overloading functions can be class members or standalone functions.

Operators as Member Functions and Global Functions

Whether an operator function is implemented as a member function or as a global function, the operator is still used the same way in expressions. So which is best?

When an operator function is implemented as a member function, the leftmost (or only) operand must be an object (or a reference to an object) of the operator’s class. If the left operand must be an object of a different class or a fundamental type, this operator function must be implemented as a global function (as we’ll do in Section 19.5 when overloading << and >> as the stream insertion and stream extraction operators, respectively). A global operator function can be made a friend of a class if that function must access private or protected members of that class directly.

Operator member functions of a specific class are called (implicitly by the compiler) only when the left operand of a binary operator is specifically an object of that class, or when the single operand of a unary operator is an object of that class.

Why Overloaded Stream Insertion and Stream Extraction Operators Are Overloaded as Global Functions

The overloaded stream insertion operator (<<) is used in an expression in which the left operand has type ostream &, as in cout << classObject. To use the operator in this manner where the right operand is an object of a user-defined class, it must be overloaded as a global function. To be a member function, operator << would have to be a member of the ostream class. This is not possible for user-defined classes, since we’re not allowed to modify C++ Standard Library classes. Similarly, the overloaded stream extraction operator (>>) is used in an expression in which the left operand has type istream &, as in cin >> classObject, and the right operand is an object of a user-defined class, so it, too, must be a global function. Also, each of these overloaded operator functions may require access to the private data members of the class object being output or input, so these overloaded operator functions can be made friend functions of the class for performance reasons.

Performance Tip 19.1

It’s possible to overload an operator as a global, non-friend function, but such a function requiring access to a class’s private or protected data would need to use set or get functions provided in that class’s public interface. The overhead of calling these functions could cause poor performance, so these functions can be inlined to improve performance.

Commutative Operators

Another reason why one might choose a global function to overload an operator is to enable the operator to be commutative. For example, suppose we have an object, number, of type long int, and an object bigInteger1, of class HugeInteger (a class in which integers may be arbitrarily large rather than being limited by the machine word size of the underlying hardware; class HugeInteger is developed in the chapter exercises). The addition operator (+) produces a temporary HugeInteger object as the sum of a HugeInteger and a long int (as in the expression bigInteger1 + number), or as the sum of a long int and a HugeInteger (as in the expression number + bigInteger1). Thus, we require the addition
operator to be commutative (exactly as it is with two fundamental-type operands). The problem is that the class object must appear on the left of the addition operator if that operator is to be overloaded as a member function. So, we overload the operator as a global function to allow the \texttt{HugeInteger} to appear on the right of the addition. The \texttt{operator+} function, which deals with the \texttt{HugeInteger} on the left, can still be a member function. The global function simply swaps its arguments and calls the member function.

### 19.5 Overloading Stream Insertion and Stream Extraction Operators

You can input and output fundamental-type data using the stream extraction operator \texttt{>>} and the stream insertion operator \texttt{<<}. The C++ class libraries overload these operators to process each fundamental type, including pointers and C-style \texttt{char *} strings. You can also overload these operators to perform input and output for your own types. The program of Figs. 19.3–19.5 overloads these operators to input and output \texttt{PhoneNumber} objects in the format “(000) 000-0000.” The program assumes telephone numbers are input correctly.

---

**Fig. 19.3** | \texttt{PhoneNumber} class with overloaded stream insertion and stream extraction operators as \texttt{friend} functions.

```cpp
1 // Fig. 19.3: PhoneNumber.h
2 // PhoneNumber class definition
3 #ifndef PHONENUMBER_H
4 #define PHONENUMBER_H
5 #include <iostream>
6 #include <string>
7 using namespace std;
8
9 class PhoneNumber
10 {
11    friend ostream &operator<<( ostream &, const PhoneNumber & );
12    friend istream &operator>>( istream &, PhoneNumber & );
13  private:
14    string areaCode; // 3-digit area code
15    string exchange; // 3-digit exchange
16    string line; // 4-digit line
17  } // end class PhoneNumber
18 #endif
```

**Fig. 19.4** | Overloaded stream insertion and stream extraction operators for class \texttt{PhoneNumber}.

(Part 1 of 2.)
// Fig. 19.5: fig19_05.cpp
// Demonstrating class PhoneNumber's overloaded stream insertion and stream extraction operators.
#include <iostream>
#include "PhoneNumber.h"
using namespace std;

int main()
{
    PhoneNumber phone; // create object phone
    cout << "Enter phone number in the form (123) 456-7890:" << endl;
    cin >> phone; // cin >> phone invokes operator>> by implicitly issuing
    // the global function call operator>>( cin, phone )
    cout << "The phone number entered was: ";
    cout << phone << endl;
    // cout << phone invokes operator<< by implicitly issuing
    // the global function call operator<<( cout, phone )
    // cout << phone << endl;
} // end main

Fig. 19.5 | Overloaded stream insertion and stream extraction operators. (Part I of 2.)
Chapter 19  Operator Overloading

The stream extraction operator function `operator>>(Fig. 19.4, lines 21–30)` takes `iostream` reference `input` and `PhoneNumber` reference `number` as arguments and returns an `iostream` reference. Operator function `operator>>` inputs phone numbers of the form

```
(800) 555-1212
```

into objects of class `PhoneNumber`. When the compiler sees the expression

```
cin >> phone
```

in line 16 of Fig. 19.5, the compiler generates the global function call

```
operator>>( cin, phone );
```

When this call executes, reference parameter `input` (Fig. 19.4, line 21) becomes an alias for `cin` and reference parameter `number` becomes an alias for `phone`. The operator function reads as strings the three parts of the telephone number into the `areaCode` (line 24), exchange (line 26) and `line` (line 28) members of the `PhoneNumber` object referenced by parameter `number`. Stream manipulator `setw` limits the number of characters read into each string. When used with `cin` and strings, `setw` restricts the number of characters read to the number of characters specified by its argument (i.e., `setw( 3 )` allows three characters to be read). The parentheses, space and dash characters are skipped by calling `iostream` member function `ignore` (Fig. 19.4, lines 23, 25 and 27), which discards the specified number of characters in the input stream (one character by default). Function `operator>>` returns `iostream` reference `input` (i.e., `cin`). This enables input operations on `PhoneNumber` objects to be cascaded with input operations on other `PhoneNumber` objects or on objects of other data types. For example, a program can input two `PhoneNumber` objects in one statement as follows:

```
cin >> phone1 >> phone2;
```

First, the expression `cin >> phone1` executes by making the global function call

```
operator>>( cin, phone1 );
```

This call then returns a reference to `cin` as the value of `cin >> phone1`, so the remaining portion of the expression is interpreted simply as `cin >> phone2`. This executes by making the global function call

```
operator>>( cin, phone2 );
```

The stream insertion operator function (Fig. 19.4, lines 11–16) takes an `ostream` reference (output) and a `const PhoneNumber` reference (number) as arguments and returns an `ostream` reference. Function `operator<<` displays objects of type `PhoneNumber`. When the compiler sees the expression

```
cout << phone
```

Fig. 19.5  Overloaded stream insertion and stream extraction operators. (Part 2 of 2.)

Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
in line 22 of Fig. 19.5, the compiler generates the global function call

```
operator<<( cout, phone );
```

Function operator<< displays the parts of the telephone number as strings, because they’re stored as string objects.

### Error-Prevention Tip 19.1

Returning a reference from an overloaded << or >> operator function is typically successful because cout, cin and most stream objects are global, or at least long-lived. Returning a reference to an automatic variable or other temporary object is dangerous—this can create “dangling references” to nonexisting objects.

The functions operator>> and operator<< are declared in PhoneNumber as global, friend functions (Fig. 19.3, lines 12–13). They’re global functions because the object of class PhoneNumber is the operator’s right operand. Remember, overloaded operator functions for binary operators can be member functions only when the left operand is an object of the class in which the function is a member. Overloaded input and output operators are declared as friends if they need to access non-public class members directly for performance reasons or because the class may not offer appropriate get functions. Also, the PhoneNumber reference in function operator<<’s parameter list (Fig. 19.4, line 11) is const, because the PhoneNumber will simply be output, and the PhoneNumber reference in function operator>>’s parameter list (line 21) is non-const, because the PhoneNumber object must be modified to store the input telephone number in the object.

### Software Engineering Observation 19.3

New input/output capabilities for user-defined types are added to C++ without modifying standard input/output library classes. This is another example of C++’s extensibility.

### 19.6 Overloading Unary Operators

A unary operator for a class can be overloaded as a non-static member function with no arguments or as a global function with one argument that must be an object (or a reference to an object) of the class. Member functions that implement overloaded operators must be non-static so that they can access the non-static data in each object of the class. Remember that static member functions can access only static members of the class.

Later in this chapter, we’ll overload unary operator ! to test whether an object of the String class we create (Section 19.11) is empty and return a bool result. Consider the expression !s, in which s is an object of class String. When a unary operator such as ! is overloaded as a member function with no arguments and the compiler sees the expression !s, the compiler generates the function call s.operator!(). The operand s is the class object for which the String class member function operator! is being invoked. The function is declared in the class definition as follows:

```cpp
class String
{
    public:
        bool operator!() const;

    ...
}; // end class String
```
A unary operator such as \(!\) may be overloaded as a global function with one parameter in two different ways—either with a parameter that is an object (this requires a copy of the object, so the side effects of the function are not applied to the original object), or with a parameter that is a reference to an object (no copy of the original object is made, so all side effects of this function are applied to the original object). If \(s\) is a String class object (or a reference to a String class object), then \(!s\) is treated as if the call operator!(s) had been written, invoking the global operator! function that is declared as follows:

```cpp
bool operator!( const String &);
```

### 19.7 Overloading Binary Operators

A binary operator can be overloaded as a non-static member function with one parameter or as a global function with two parameters (one of those parameters must be either a class object or a reference to a class object).

Later in this chapter, we’ll overload < to compare two String objects. When overloading binary operator < as a non-static member function of a String class with one argument, if \(y\) and \(z\) are String-class objects, then \(y < z\) is treated as if \(y\.\text{operator}<(z)\) had been written, invoking the operator< member function declared below:

```cpp
class String
{
public:
   bool operator<( const String & ) const;
}; // end class String
```

As a global function, binary operator < must take two arguments—one of which must be an object (or a reference to an object) of the class. If \(y\) and \(z\) are String-class objects or references to String-class objects, then \(y < z\) is treated as if the call operator<(y, z) had been written in the program, invoking global-function operator< declared as follows:

```cpp
bool operator<( const String &, const String &);
```

### 19.8 Dynamic Memory Management

A standard C++ array data structure is fixed in size once it’s created. The size is specified with a constant at compile time. Sometimes it’s useful to determine the size of an array dynamically at execution time and then create the array. C++ enables you to control the allocation and deallocation of memory in a program for objects and for arrays of any built-in or user-defined type. This is known as dynamic memory management and is performed with the operators \texttt{new} and \texttt{delete}.

You can use the new operator to dynamically allocate (i.e., reserve) the exact amount of memory required to hold an object or array at execution time. The object or array is created in the free store (also called the heap)—a region of memory assigned to each program for storing dynamically allocated objects. Once memory is allocated in the free store, you can access it via the pointer that operator new returns. When you no longer need the memory, you can return it to the free store by using the delete operator to deallocate (i.e., release) the memory, which can then be reused by future new operations.
19.8  Dynamic Memory Management

Obtaining Dynamic Memory with new

Let’s discuss the details of using the new and delete operators to dynamically allocate memory to store objects, fundamental types and arrays. Consider the following statement:

```
Time *timePtr = new Time;
```

The new operator allocates storage of the proper size for an object of type Time, calls the default constructor to initialize the object and returns a pointer to the type specified to the right of the new operator (i.e., a Time*). If new is unable to find sufficient space in memory for the object, it indicates that an error occurred by “throwing an exception.” Chapter 24, Exception Handling, discusses how to deal with new failures. In particular, we’ll show how to “catch” the exception thrown by new and deal with it. When a program does not “catch” an exception, the program terminates immediately.

Releasing Dynamic Memory with delete

To destroy a dynamically allocated object and free the space for the object, use the delete operator as follows:

```
delete timePtr;
```

This statement first calls the destructor for the object to which timePtr points, then deallocates the memory associated with the object, returning the memory to the free store.

Initializing Dynamic Memory

You can provide an initializer for a newly created fundamental-type variable, as in

```
double *ptr = new double(3.14159);
```

which initializes a newly created double to 3.14159 and assigns the resulting pointer to ptr. The same syntax can be used to specify a comma-separated list of arguments to the constructor of an object. For example,

```
Time *timePtr = new Time(12, 45, 0);
```

initializes a new Time object to 12:45 PM and assigns the resulting pointer to timePtr.

Dynamically Allocating Arrays with new []

You can also use the new operator to allocate arrays dynamically. For example, a 10-element integer array can be allocated and assigned to gradesArray as follows:

```
int *gradesArray = new int[10];
```

which declares int pointer gradesArray and assigns to it a pointer to the first element of a dynamically allocated 10-element array of ints. The size of an array created at compile time must be specified using a constant integral expression; however, a dynamically allocated array’s size can be specified using any non-negative integral expression that can be evaluated at execution time. Also, when allocating an array of objects dynamically, you cannot pass arguments to each object’s constructor—each object is initialized by its default constructor. For fundamental types, the elements are initialized to 0 or the equivalent of 0.
(e.g., chars are initialized to the null character, '\0'). Although an array name is a pointer to the array’s first element, the following is not allowed for dynamically allocated memory:

```c
int gradesArray[] = new int[10];
```

### Releasing Dynamically Allocated Arrays with `delete []`
To deallocate the memory to which `gradesArray` points, use the statement

```c
delete [] gradesArray;
```

If the pointer points to an array of objects, the statement first calls the destructor for every object in the array, then deallocates the memory. If the preceding statement did not include the square brackets (`[]`) and `gradesArray` pointed to an array of objects, the result is undefined. Some compilers call the destructor only for the first object in the array. Using `delete` on a null pointer (i.e., a pointer with the value 0) has no effect.

#### Common Programming Error 19.6
Using `delete` instead of `delete []` for arrays of objects can lead to runtime logic errors. To ensure that every object in the array receives a destructor call, always delete memory allocated as an array with operator `delete []`. Similarly, always delete memory allocated as an individual element with operator `delete`—the result of deleting a single object with operator `delete []` is undefined.

### 19.9 Case Study: Array Class
We discussed arrays in Chapter 6. An array is not much more than a pointer to some space in memory. Pointer-based arrays have many problems, including:

- A program can easily “walk off” either end of an array, because C++ does not check whether subscripts fall outside the range of an array (though you can still do this explicitly).
- Arrays of size `n` must number their elements `0, …, n – 1`; alternate subscript ranges are not allowed.
- An entire array cannot be input or output at once; each array element must be read or written individually (unless the array is a null-terminated C string).
- Two arrays cannot be meaningfully compared with equality or relational operators (because the array names are simply pointers to where the arrays begin in memory and two arrays will always be at different memory locations).
- When an array is passed to a general-purpose function designed to handle arrays of any size, the array’s size must be passed as an additional argument.
- One array cannot be assigned to another with the assignment operator(s) (because array names are `const` pointers and a `const` pointer cannot be used on the left side of an assignment operator).

C++ provides the means to implement more robust array capabilities via classes and operator overloading. You can develop an array class that is preferable to “raw” arrays. In this example, we create a powerful `Array` class that performs range checking to ensure that
subscripts remain within the bounds of the Array. The class allows one array object to be assigned to another with the assignment operator. Array objects know their size, so the size does not need to be passed separately to functions that receive Array parameters. Entire Arrays can be input or output with the stream extraction and stream insertion operators, respectively. You can compare Arrays with the equality operators == and !=. C++ Standard Library class template vector provides many of these capabilities as well.

This example will sharpen your appreciation of data abstraction. Class development is an interesting, creative and intellectually challenging activity—always with the goal of “crafting valuable classes.” The program of Figs. 19.6–19.8 demonstrates class Array and its overloaded operators. First we walk through main (Fig. 19.8), then we consider the class definition (Fig. 19.6) and each of its member-function definitions (Fig. 19.7).

---

### Fig. 19.6: Array.h

```cpp
// Fig. 19.6: Array.h
// Array class definition with overloaded operators.
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
using namespace std;

class Array {
public:
    Array(int = 10); // default constructor
    Array(const Array &); // copy constructor
    ~Array(); // destructor
    int getSize() const; // return size

    // subscript operator for non-const objects returns modifiable lvalue
    int &operator[](int);

    // subscript operator for const objects returns rvalue
    int operator[](int) const;

private:
    int size; // pointer-based array size
    int *ptr; // pointer to first element of pointer-based array
}; // end class Array

friend ostream &operator<<( ostream &, const Array & );
friend istream &operator>>( istream &, Array & );

const Array &operator=( const Array & ); // assignment operator
bool operator==( const Array & ) const; // equality operator

// inequality operator; returns opposite of == operator
bool operator!=( const Array &right ) const
{
    return ! ( *this == right ); // invokes Array::operator==
}

// subscript operator for non-const objects returns modifiable lvalue
int &operator[]( int );

// subscript operator for const objects returns rvalue
int operator[]( int ) const;
#endif
```

---
// Fig 19.7: Array.cpp
// Array class member- and friend-function definitions.
#include <iostream>
#include <iomanip>
#include <cstdlib> // exit function prototype
#include "Array.h" // Array class definition
using namespace std;

// default constructor for class Array (default size 10)
Array::Array(int arraySize)
{
    size = (arraySize > 0 ? arraySize : 10); // validate arraySize
    ptr = new int[size]; // create space for pointer-based array
    for (int i = 0; i < size; i++)
        ptr[i] = 0; // set pointer-based array element
} // end Array default constructor

// copy constructor for class Array;
// must receive a reference to prevent infinite recursion
Array::Array(const Array &arrayToCopy)
    : size(arrayToCopy.size)
{
    ptr = new int[size]; // create space for pointer-based array
    for (int i = 0; i < size; i++)
        ptr[i] = arrayToCopy.ptr[i]; // copy into object
} // end Array copy constructor

// destructor for class Array
Array::~Array()
{
    delete[] ptr; // release pointer-based array space
} // end destructor

// return number of elements of Array
int Array::getSize() const
{
    return size; // number of elements in Array
} // end function getSize

// overloaded assignment operator;
// const return avoids: (a1 = a2) = a3
const Array &Array::operator=(const Array &right)
{
    if (&right != this) // avoid self-assignment
    {
        // for Arrays of different sizes, deallocate original
        // left-side array, then allocate new left-side array
        if (size != right.size)
        {
            delete[] ptr; // release space
            size = right.size; // resize this object
        }
    }
    return *this;
} // end operator="
ptr = new int[size]; // create space for array copy

for (int i = 0; i < size; i++)
    ptr[i] = right.ptr[i]; // copy array into object

return *this; // enables x = y = z, for example

// determine if two Arrays are equal and
// return true, otherwise return false
bool Array::operator==(const Array &right) const
{
    if (size != right.size)
        return false; // arrays of different number of elements

    for (int i = 0; i < size; i++)
        if (ptr[i] != right.ptr[i])
            return false; // Array contents are not equal

    return true; // Arrays are equal
}

// overloaded subscript operator for non-const Arrays;
// reference return creates a modifiable lvalue
int &Array::operator[](int subscript)
{
    // check for subscript out-of-range error
    if (subscript < 0 || subscript >= size)
    {
        cerr << "Error: Subscript " << subscript << " out of range" << endl;
        exit(1); // terminate program; subscript out of range
    }

    return ptr[subscript]; // reference return
}

// overloaded subscript operator for const Arrays
// const reference return creates an rvalue
int Array::operator[](int subscript) const
{
    // check for subscript out-of-range error
    if (subscript < 0 || subscript >= size)
    {
        cerr << "Error: Subscript " << subscript << " out of range" << endl;
        exit(1); // terminate program; subscript out of range
    }

    return ptr[subscript]; // returns copy of this element
}
Chapter 19  Operator Overloading

// overloaded input operator for class Array;
// inputs values for entire Array
istream &operator>>( istream &input, Array &a )
{
    for ( int i = 0; i < a.size; i++ )
        input >> a.ptr[i];
    return input; // enables cin >> x >> y;
}

// overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
    int i;

    // output private ptr-based array
    for ( i = 0; i < a.size; i++ )
    {
        output << setw( 12 ) << a.ptr[i];
        if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
            output << endl;
    } // end for

    if ( i % 4 != 0 ) // end last line of output
        output << endl;

    return output; // enables cout << x << y;
}

Fig. 19.8  |  Array class test program. (Part 1 of 3.)

Fig. 19.7  |  Array class member- and friend-function definitions. (Part 3 of 3.)
Fig. 19.8 | Array class test program. (Part 2 of 3.)
Size of Array integers1 is 7
Array after initialization:
0 0 0 0
0 0 0

Size of Array integers2 is 10
Array after initialization:
0 0 0 0
0 0 0 0
0 0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the Arrays contain:
integers1:
1 2 3 4
5 6 7
integers2:
8 9 10 11
12 13 14 15
16 17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of Array integers3 is 7
Array after initialization:
1 2 3 4
5 6 7

Assigning integers2 to integers1:
integers1:
8 9 10 11
12 13 14 15
16 17
integers2:
8 9 10 11
12 13 14 15
16 17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
8 9 10 11
12 1000 14 15
16 17

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range

Fig. 19.8 | Array class test program. (Part 3 of 3.)
Creating Arrays, Outputting Their Size and Displaying Their Contents
The program begins by instantiating two objects of class Array—integers1 (Fig. 19.8, line 9) with seven elements, and integers2 (Fig. 19.8, line 10) with the default Array size—10 elements (specified by the Array default constructor’s prototype in Fig. 19.6, line 14). Lines 13–15 use member function getSize to determine the size of integers1 and output integers1, using the Array overloaded stream insertion operator. The sample output confirms that the Array elements were set correctly to zeros by the constructor. Next, lines 18–20 output the size of Array integers2 and output integers2, using the Array overloaded stream insertion operator.

Using the Overloaded Stream Insertion Operator to Fill an Array
Line 23 prompts the user to input 17 integers. Line 24 uses the Array overloaded stream extraction operator to read these values into both arrays. The first seven values are stored in integers1 and the remaining 10 values are stored in integers2. Lines 26–28 output the two arrays with the overloaded Array stream insertion operator to confirm that the input was performed correctly.

Using the Overloaded Inequality Operator
Line 33 tests the overloaded inequality operator by evaluating the condition

\[\text{integers1} \neq \text{integers2}\]

The program output shows that the Arrays are not equal.

Initializing a New Array with a Copy of an Existing Array’s Contents
Line 38 instantiates a third Array called integers3 and initializes it with a copy of Array integers1. This invokes the Array copy constructor to copy the elements of integers1 into integers3. We discuss the details of the copy constructor shortly. The copy constructor can also be invoked by writing line 38 as follows:

\[
\text{Array integers3} = \text{integers1};
\]

The equal sign in the preceding statement is not the assignment operator. When an equal sign appears in the declaration of an object, it invokes a constructor for that object. This form can be used to pass only a single argument to a constructor.

Lines 40–42 output the size of integers3 and output integers3, using the Array overloaded stream insertion operator to confirm that the Array elements were set correctly by the copy constructor.

Using the Overloaded Assignment Operator
Next, line 46 tests the overloaded assignment operator (=) by assigning integers2 to integers1. Lines 48–49 print both Array objects to confirm that the assignment was successful. Note that integers1 originally held 7 integers and was resized to hold a copy of the 10 elements in integers2. As we’ll see, the overloaded assignment operator performs this resizing operation in a manner that is transparent to the client code.

Using the Overloaded Equality Operator
Next, line 54 uses the overloaded equality operator (==) to confirm that objects integers1 and integers2 are indeed identical after the assignment.
Using the Overloaded Subscript Operator

Line 58 uses the overloaded subscript operator to refer to integers1[5]—an in-range element of integers1. This subscripted name is used as an rvalue to print the value stored in integers1[5]. Line 62 uses integers1[5] as a modifiable lvalue on the left side of an assignment statement to assign a new value, 1000, to element 5 of integers1. We’ll see that operator[] returns a reference to use as the modifiable lvalue after the operator confirms that 5 is a valid subscript for integers1.

Line 67 attempts to assign the value 1000 to integers1[15]—an out-of-range element. In this example, operator[] determines that the subscript is out of range, prints a message and terminates the program. We highlighted line 67 of the program in darker blue to emphasize that it’s an error to access an element that is out of range. This is a runtime logic error.

Interestingly, the array subscript operator [] is not restricted for use only with arrays; it also can be used, for example, to select elements from other kinds of container classes, such as linked lists, strings and dictionaries. Also, when operator[] functions are defined, subscripts no longer have to be integers—characters, strings, floats or even objects of user-defined classes also could be used.

Array Class Definition

Now that we’ve seen how this program operates, let’s walk through the class header (Fig. 19.6). As we refer to each member function in the header, we discuss that function’s implementation in Fig. 19.7. In Fig. 19.6, lines 34–35 represent the private data members of class Array. Each Array object consists of a size member indicating the number of elements in the Array and an int pointer—ptr—that points to the dynamically allocated pointer-based array of integers managed by the Array object.

Overloading the Stream Insertion and Stream Extraction Operators as friends

Lines 11–12 of Fig. 19.6 declare the overloaded stream insertion operator and the overloaded stream extraction operator to be friends of class Array. When the compiler sees an expression like cout << arrayObject, it invokes global function operator<< with the call

```cpp
operator<<( cout, arrayObject )
```

When the compiler sees an expression like cin >> arrayObject, it invokes global function operator>> with the call

```cpp
operator>>( cin, arrayObject )
```

We note again that these stream insertion and stream extraction operator functions cannot be members of class Array, because the Array object is always mentioned on the right side of the stream insertion operator and the stream extraction operator. If these operator functions were to be members of class Array, the following awkward statements would have to be used to output and input an Array:

```cpp
arrayObject << cout;  
arrayObject >> cin;
```

Such statements would be confusing to most C++ programmers, who are familiar with cout and cin appearing as the left operands of << and >>, respectively.
Function operator<< (defined in Fig. 19.7, lines 119–136) prints the number of elements indicated by size from the integer array to which ptr points. Function operator>>(defined in Fig. 19.7, lines 110–116) inputs directly into the array to which ptr points. Each of these operator functions returns an appropriate reference to enable cascaded output or input statements, respectively. Each of these functions has access to an Array’s private data because these functions are declared as friends of class Array. Also, class Array’s getSize and operator[] functions could be used by operator<< and operator>>, in which case these operator functions would not need to be friends of class Array. However, the additional function calls might increase execution-time overhead.

**Array Default Constructor**

Line 14 of Fig. 19.6 declares the default constructor for the class and specifies a default size of 10 elements. When the compiler sees a declaration like line 10 in Fig. 19.8, it invokes class Array’s default constructor (remember that the default constructor in this example actually receives a single int argument that has a default value of 10). The default constructor (defined in Fig. 19.7, lines 10–17) validates and assigns the argument to data member size, uses new to obtain the memory for the internal pointer-based representation of this array and assigns the pointer returned by new to data member ptr. Then the constructor uses a for statement to set all the elements of the array to zero. It’s possible to have an Array class that does not initialize its members if, for example, these members are to be read at some later time; but this is considered to be a poor programming practice. Arrays, and objects in general, should be properly initialized and maintained in a consistent state.

**Array Copy Constructor**

Line 15 of Fig. 19.6 declares a copy constructor (defined in Fig. 19.7, lines 21–28) that initializes an Array by making a copy of an existing Array object. Such copying must be done carefully to avoid the pitfall of leaving both Array objects pointing to the same dynamically allocated memory. This is exactly the problem that would occur with default memberwise copying, if the compiler is allowed to define a default copy constructor for this class. Copy constructors are invoked whenever a copy of an object is needed, such as in passing an object by value to a function, returning an object by value from a function or initializing an object with a copy of another object of the same class. The copy constructor is called in a declaration when an object of class Array is instantiated and initialized with another object of class Array, as in the declaration in line 38 of Fig. 19.8.

---

**Software Engineering Observation 19.4**

The argument to a copy constructor should be a const reference to allow a const object to be copied.

**Common Programming Error 19.7**

A copy constructor must receive its argument by reference, not by value. Otherwise, the copy constructor call results in infinite recursion (a fatal logic error) because receiving an object by value requires the copy constructor to make a copy of the argument object. Recall that any time a copy of an object is required, the class’s copy constructor is called. If the copy constructor received its argument by value, the copy constructor would call itself recursively to make a copy of its argument!
The copy constructor for `Array` uses a member initializer (Fig. 19.7, line 22) to copy the size of the initializer `Array` into data member `size`, uses `new` (line 24) to obtain the memory for the internal pointer-based representation of this `Array` and assigns the pointer returned by `new` to data member `ptr`. Then the copy constructor uses a `for` statement to copy all the elements of the initializer `Array` into the new `Array` object. An object of a class can look at the private data of any other object of that class (using a handle that indicates which object to access).

**Common Programming Error 19.8**

If the copy constructor simply copied the pointer in the source object to the target object’s pointer, then both objects would point to the same dynamically allocated memory. The first destructor to execute would then delete the dynamically allocated memory, and the other object’s `ptr` would be undefined, a situation called a **dangling pointer**—this would likely result in a serious run-time error (such as early program termination) when the pointer was used.

**Array Destructor**

Line 16 of Fig. 19.6 declares the class’s destructor (defined in Fig. 19.7, lines 31–34). The destructor is invoked when an object of class `Array` goes out of scope. The destructor uses `delete []` to release the memory allocated dynamically by `new` in the constructor.

**Error-Prevention Tip 19.2**

If after deleting dynamically allocated memory, the pointer will continue to exist in memory, set the pointer’s value to 0 to indicate that the pointer no longer points to memory in the free store. By setting the pointer to 0, the program loses access to that free-store space, which could be reallocated for a different purpose. If you do not set the pointer to 0, your code could inadvertently access the reallocated memory, causing subtle, nonrepeatable logic errors.

**getSize Member Function**

Line 17 of Fig. 19.6 declares function `getSize` (defined in Fig. 19.7, lines 37–40) that returns the number of elements in the `Array`.

**Overloaded Assignment Operator**

Line 19 of Fig. 19.6 declares the overloaded assignment operator function for the class. When the compiler sees the expression `integers1 = integers2` in line 46 of Fig. 19.8, the compiler invokes member function `operator=` with the call

```
integers1.operator=( integers2 )
```

Member function `operator=’s` implementation (Fig. 19.7, lines 44–62) tests for **self-assignment** (line 46) in which an `Array` object is being assigned to itself. When `this` is equal to the right operand’s address, a self-assignment is being attempted, so the assignment is skipped (i.e., the object already is itself; in a moment we’ll see why self-assignment is dangerous). If it isn’t a self-assignment, then the function determines whether the sizes of the

---

1. Operator `new` could fail to obtain the needed memory. We deal with `new` failures in Chapter 24.
two arrays are identical (line 50); in that case, the original array of integers in the left-side Array object is not reallocated. Otherwise, operator= uses delete (line 52) to release the memory originally allocated to the target array, copies the size of the source array to the size of the target array (line 53), uses new to allocate memory for the target array and places the pointer returned by new into the array’s ptr member. Then the for statement in lines 57–58 copies the array elements from the source array to the target array. Regardless of whether this is a self-assignment, the member function returns the current object (i.e., *this in line 61) as a constant reference; this enables cascaded Array assignments such as x = y = z, but prevents ones like (x = y) = z because z cannot be assigned to the const Array reference that is returned by (x = y). If self-assignment occurs, and function operator= did not test for this case, operator= would unnecessarily copy the elements of the Array into itself.

**Overloaded Equality and Inequality Operators**

Line 20 of Fig. 19.6 declares the overloaded equality operator (==) for the class. When the compiler sees the expression integers1 == integers2 in line 54 of Fig. 19.8, the compiler invokes member function operator== with the call

```
integers1.operator==( integers2 )
```

Member function operator== (defined in Fig. 19.7, lines 66–76) immediately returns false if the size members of the arrays are not equal. Otherwise, operator== compares each pair of elements. If they’re all equal, the function returns true. The first pair of elements to differ causes the function to return false immediately.

Lines 23–26 of the header file define the overloaded inequality operator (!=) for the class. Member function operator!= uses the overloaded operator== function to determine whether one Array is equal to another, then returns the opposite of that result. Writing operator!= in this manner enables you to reuse operator==, which reduces the amount of code that must be written in the class. Also, the full function definition for operator!= is in the Array header file. This allows the compiler to inline the definition of operator!= to eliminate the overhead of the extra function call.

**Software Engineering Observation 19.5**

A copy constructor, a destructor and an overloaded assignment operator are usually provided as a group for any class that uses dynamically allocated memory.

**Common Programming Error 19.9**

Not providing an overloaded assignment operator and a copy constructor for a class when objects of that class contain pointers to dynamically allocated memory is a logic error.

**Software Engineering Observation 19.6**

It’s possible to prevent one object of a class from being assigned to another. This is done by declaring the assignment operator as a private member of the class.

**Software Engineering Observation 19.7**

It’s possible to prevent class objects from being copied; to do this, simply make both the overloaded assignment operator and the copy constructor of that class private.
Overloaded Subscript Operators

Lines 29 and 32 of Fig. 19.6 declare two overloaded subscript operators (defined in Fig. 19.7 in lines 80–91 and 95–106, respectively). When the compiler sees the expression `integers1[5]` (Fig. 19.8, line 58), it invokes the appropriate overloaded `operator[]` member function by generating the call

```
integers1.operator[]( 5 )
```

The compiler creates a call to the `const` version of `operator[]` (Fig. 19.7, lines 95–106) when the subscript operator is used on a `const Array` object. For example, if `const` object `z` is instantiated with the statement

```
const Array z( 5 );
```

then the `const` version of `operator[]` is required to execute a statement such as

```
cout << z[ 3 ] << endl;
```

Remember, a program can invoke only the `const` member functions of a `const` object.

Each definition of `operator[]` determines whether the subscript it receives as an argument is in range. If it isn’t, each function prints an error message and terminates the program with a call to function `exit` (header `<cstdlib>`). If the subscript is in range, the `non-const` version of `operator[]` returns the appropriate array element as a reference so that it may be used as a modifiable `lvalue` (e.g., on the left side of an assignment statement). If the subscript is in range, the `const` version of `operator[]` returns a copy of the appropriate element of the array. The returned character is an `rvalue`.

19.10 Converting between Types

Most programs process information of many types. Sometimes all the operations “stay within a type.” For example, adding an `int` to an `int` produces an `int`. It’s often necessary, however, to convert data of one type to data of another type. This can happen in assignments, in calculations, in passing values to functions and in returning values from functions. The compiler knows how to perform certain conversions among fundamental types. You can use cast operators to force conversions among fundamental types.

But what about user-defined types? The compiler cannot know in advance how to convert among user-defined types, and between user-defined types and fundamental types, so you must specify how to do this. Such conversions can be performed with `conversion constructors`—single-argument constructors that turn objects of other types (including fundamental types) into objects of a particular class.

A `conversion operator` (also called a cast operator) can be used to convert an object of one class into an object of another class or into an object of a fundamental type. Such a conversion operator must be a non-static member function. The function prototype

```
A::operator char *() const;
```
19.11 Building a String Class

Our String class provides a conversion constructor that takes a const char * argument and initializes a String object containing that same character string. Recall that any single-argument constructor can be thought of as a conversion constructor. Such constructors are helpful when we’re doing any String operation using char * arguments. The conversion constructor can convert a char * string into a String object, which can then be assigned to the target String object. The availability of this conversion constructor means that it
isn’t necessary to supply an overloaded assignment operator for assigning character strings
to String objects. When the compiler encounters the statement

```cpp
myString = "hello";
```

where `myString` is a String object, the compiler invokes the conversion constructor to
create a temporary String object containing the character string "hello"; then class
String’s overloaded assignment operator is invoked to assign the temporary String object
to String object `myString`.

**Software Engineering Observation 19.8**

When a conversion constructor is used to perform an implicit conversion, C++ can apply
only one implicit constructor call (i.e., a single user-defined conversion) to try to match the
needs of another overloaded operator. The compiler will not satisfy an overloaded
operator’s needs by performing a series of implicit, user-defined conversions.

The String conversion constructor could be invoked in a declaration such as

```cpp
String s1("happy");
```

It can also be invoked when you pass a C string to a function that expects a String argu-
ment or when you return a C string from a function with a String return type.

**Overloaded Unary Negation Operator**

The overloaded negation operator determines whether a String object is empty. For ex-
ample, when the compiler sees the expression `!string1`, it generates the function call

```cpp
string1.operator!()
```

This function returns true if the String’s length is equal to zero, and false otherwise.

**Overloaded Function Call Operator**

Overloading the function call operator `()` is powerful, because functions can take an ar-
bitrary number of parameters. In class String, we overload this operator to select a sub-
string from a String. The operator’s two integer parameters specify the start location and
the length of the substring to be selected. If the start location is out of range or the sub-
string length is negative, the operator simply returns an empty String. If the substring
length is 0, then the substring is selected to the end of the String object. Suppose `string1`
is a String object containing the string "AEIOU". When the compiler encounters the ex-
pression `string1(2, 2)`, it generates the member-function call

```cpp
string1.operator()(2, 2)
```

which returns a String containing "IO".

**19.12 Overloading ++ and --**

The prefix and postfix versions of the increment and decrement operators can all be over-
loaded. We’ll see how the compiler distinguishes between the prefix version and the post-
fix version of an increment or decrement operator.

To overload the increment operator to allow both prefix and postfix increment usage,
each overloaded operator function must have a distinct signature, so that the compiler will
be able to determine which version of \texttt{++} is intended. The prefix versions are overloaded exactly as any other prefix unary operator would be.

\textit{Overloading the Prefix Increment Operator}

Suppose, for example, that we want to add 1 to the day in Date object \texttt{d1}. When the compiler sees the preincrementing expression \texttt{++d1}, the compiler generates the member-function call

\begin{verbatim}
d1.operator++()
\end{verbatim}

The prototype for this operator function would be

\begin{verbatim}
Date &operator++();
\end{verbatim}

If the prefix increment operator is implemented as a global function, then, when the compiler sees the expression \texttt{++d1}, the compiler generates the function call

\begin{verbatim}
operator++( d1 )
\end{verbatim}

The prototype for this operator function would be declared in the Date class as

\begin{verbatim}
Date &operator++( Date & );
\end{verbatim}

\textit{Overloading the Postfix Increment Operator}

Overloading the postfix increment operator presents a challenge, because the compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions. The \textit{convention} that has been adopted in C++ is that, when the compiler sees the postincrementing expression \texttt{d1++}, it generates the member-function call

\begin{verbatim}
d1.operator++( 0 )
\end{verbatim}

The prototype for this function is

\begin{verbatim}
Date operator++( int )
\end{verbatim}

The argument 0 is strictly a “dummy value” that enables the compiler to distinguish between the prefix and postfix increment operator functions. The same syntax is used to differentiate between the prefix and postfix decrement operator functions.

If the postfix increment is implemented as a global function, then, when the compiler sees the expression \texttt{d1++}, the compiler generates the function call

\begin{verbatim}
operator++( d1, 0 )
\end{verbatim}

The prototype for this function would be

\begin{verbatim}
Date operator++( Date & , int );
\end{verbatim}

Once again, the 0 argument is used by the compiler to distinguish between the prefix and postfix increment operators implemented as global functions. Note that the postfix increment operator returns Date objects by value, whereas the prefix increment operator returns Date objects by reference, because the postfix increment operator typically returns a temporary object that contains the original value of the object before the increment occurred. C++ treats such objects as \textit{rvalues}, which cannot be used on the left side of an assignment. The prefix increment operator returns the actual incremented object with its new value. Such an object can be used as an \textit{lvalue} in a continuing expression.
Everything stated in this section for overloading prefix and postfix increment operators applies to overloading predecrement and postdecrement operators. Next, we examine a `Date` class with overloaded prefix and postfix increment operators.

### 19.13 Case Study: A Date Class

The program of Figs. 19.9–19.11 demonstrates a `Date` class, which uses overloaded prefix and postfix increment operators to add 1 to the day in a `Date` object, while causing appropriate increments to the month and year if necessary. The `Date` header file (Fig. 19.9) specifies that `Date`'s public interface includes an overloaded stream insertion operator (line 11), a default constructor (line 13), a `setDate` function (line 14), an overloaded prefix increment operator (line 15), an overloaded postfix increment operator (line 16), an overloaded `+=` addition assignment operator (line 17), a function to test for leap years (line 18) and a function to determine whether a day is the last day of the month (line 19).

```cpp
// Fig. 19.9: Date.h
// Date class definition with overloaded increment operators.
#ifndef DATE_H
#define DATE_H

#include <iostream>
using namespace std;

class Date
{
    friend ostream &operator<<( ostream &, const Date & );

public:
    Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
    void setDate( int, int, int ); // set month, day, year
    Date &operator++(); // prefix increment operator
    Date operator++( int ); // postfix increment operator
    const Date &operator+=( int ); // add days, modify object
    static bool leapYear( int ); // is date in a leap year?
    bool endOfMonth( int ) const; // is date at the end of month?

private:
    int month;
    int day;
    int year;
    static const int days[]; // array of days per month
    void helpIncrement(); // utility function for incrementing date
}; // end class Date

#endif
```

**Fig. 19.9**  Date class definition with overloaded increment operators.
// Fig. 19.10: Date.cpp
// Date class member- and friend-function definitions.
#include <iostream>
#include <string>
#include "Date.h"
using namespace std;

// initialize static member; one classwide copy
const int Date::days[] = {
    0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};

// Date constructor
Date::Date( int m, int d, int y )
{
    setDate( m, d, y );
} // end Date constructor

// set month, day and year
void Date::setDate( int mm, int dd, int yy )
{
    month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
    year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;

    // test for a leap year
    if ( month == 2 && leapYear( year ) )
        day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
    else
        day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
} // end function setDate

// overloaded prefix increment operator
Date &Date::operator++()
{
    helpIncrement(); // increment date
    return *this; // reference return to create an lvalue
} // end function operator++

// overloaded postfix increment operator; note that the
dummy integer parameter does not have a parameter name
Date Date::operator++( int )
{
    Date temp = *this; // hold current state of object
    helpIncrement();

    // return unincremented, saved, temporary object
    return temp; // value return; not a reference return
} // end function operator++

// add specified number of days to date
const Date &Date::operator+=( int additionalDays )
{
    for ( int i = 0; i < additionalDays; i++ )
        helpIncrement();
} // end function operator+=
Fig. 19.10 | Date class member- and friend-function definitions. (Part 2 of 2.)
```cpp
// Fig. 19.11: fig19_11.cpp
// Date class test program.
#include <iostream>
#include "Date.h" // Date class definition
using namespace std;

int main()
{
    Date d1; // defaults to January 1, 1900
    Date d2( 12, 27, 1992 ); // December 27, 1992
    Date d3( 0, 99, 8045 ); // invalid date

    cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
    d2 += 7; // December 30, 1992
    cout << "d2 += 7 is " << d2;
    d3.setDate( 2, 28, 1992 );
    cout << "\n\nd3 is " << d3;
    ++d3; // February 29, 1992 (leap year allows 29th)

    Date d4( 7, 13, 2002 );

    cout << "\n\nTesting the prefix increment operator:
" " d4 is " << d4 << endl;
    ++d4; // July 14, 2002
    cout << "d4 is " << d4 << endl;
    d4++; // July 15, 2002
    cout << "d4 is " << d4 << endl;

    d1 is January 1, 1900
    d2 is December 27, 1992
    d3 is January 1, 1900
    d2 += 7 is January 3, 1993
    d3 is February 28, 1992
    ++d3 is February 29, 1992 (leap year allows 29th)

    Testing the prefix increment operator:
    d4 is July 13, 2002
    ++d4 is July 14, 2002
    d4 is July 14, 2002

    Testing the postfix increment operator:
    d4 is July 14, 2002
    d4++ is July 14, 2002
    d4 is July 15, 2002
```

Fig. 19.11 | Date class test program.
Function main (Fig. 19.11) creates three Date objects (lines 9–11)—d1 is initialized by default to January 1, 1900; d2 is initialized to December 27, 1992; and d3 is initialized to an invalid date. The Date constructor (defined in Fig. 19.10, lines 13–16) calls setDate to validate the month, day and year specified. An invalid month is set to 1, an invalid year is set to 1900 and an invalid day is set to 1.

Line 13 of main output each of the constructed Date objects, using the overloaded stream insertion operator (defined in Fig. 19.10, lines 98–105). Line 14 of main uses the overloaded operator += to add seven days to d2. Line 16 uses function setDate to set d3 to February 28, 1992, which is a leap year. Then, line 18 preincrements d3 to show that the date increments properly to February 29. Next, line 20 creates a Date object, d4, which is initialized with the date July 13, 2002. Then line 24 increments d4 by 1 with the overloaded prefix increment operator. Lines 22–25 output d4 before and after the preincrement operation to confirm that it worked correctly. Finally, line 29 increments d4 with the overloaded postfix increment operator. Lines 27–30 output d4 before and after the postincrement operation to confirm that it worked correctly.

Overloading the prefix increment operator is straightforward. The prefix increment operator (defined in Fig. 19.10, lines 32–36) calls utility function helpIncrement (defined in Fig. 19.10, lines 78–95) to increment the date. This function deals with “wraparounds” or “carries” that occur when we increment the last day of the month. These carries require incrementing the month. If the month is already 12, then the year must also be incremented and the month must be set to 1. Function helpIncrement uses function endOfMonth to increment the day correctly.

The overloaded prefix increment operator returns a reference to the current Date object (i.e., the one that was just incremented). This occurs because the current object, *this, is returned as a Date &. This enables a preincremented Date object to be used as an lvalue, which is how the built-in prefix increment operator works for fundamental types.

Overloading the postfix increment operator (defined in Fig. 19.10, lines 40–47) is trickier. To emulate the effect of the postincrement, we must return an unincremented copy of the Date object. For example, if int variable x has the value 7, the statement

```cpp
cout << x++ << endl;
```

outputs the original value of variable x. So we’d like our postfix increment operator to operate the same way on a Date object. On entry to operator++, we save the current object (*this) in temp (line 42). Next, we call helpIncrement to increment the current Date object. Then, line 46 returns the unincremented copy of the object previously stored in temp. This function cannot return a reference to the local Date object temp, because a local variable is destroyed when the function in which it’s declared exits. Thus, declaring the return type to this function as Date & would return a reference to an object that no longer exists. Returning a reference (or a pointer) to a local variable is a common error for which most compilers will issue a warning.

### 19.14 Standard Library Class string

Building useful, reusable classes such as Array (Figs. 19.6–19.8) takes work. However, once such classes are tested and debugged, they can be reused by you, your colleagues, your company, many companies, an entire industry or even many industries (if they’re placed
in public or for-sale libraries). The designers of C++ did exactly that, building class string and class template vector into standard C++. These classes are available to anyone building applications with C++.

Figure 19.12 demonstrates many of class string’s overloaded operators, its conversion constructor for C strings and several other useful member functions, including empty, substr and at. Function empty determines whether a string is empty, function substr returns a string that represents a portion of an existing string and function at returns the character at a specific index in a string (after checking that the index is in range).

```cpp
// Fig. 19.12: fig19_12.cpp
// Standard Library string class test program.
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s1( "happy" );
    string s2( " birthday" );
    string s3;

    // test overloaded equality and relational operators
    cout << "s1 is " "" "s1 " "s2 is " "" "s2
    << "" "s3 is " "" "s3 << ""
    << "\n\nThe results of comparing s2 and s1:
"ns2 == s1 yields " "s2 == s1 ? "true" : "false"
    "ns2 != s1 yields " "s2 != s1 ? "true" : "false"
    "ns2 > s1 yields " "s2 > s1 ? "true" : "false"
    "ns2 < s1 yields " "s2 < s1 ? "true" : "false"
    "ns2 >= s1 yields " "s2 >= s1 ? "true" : "false"
    "ns2 <= s1 yields " "s2 <= s1 ? "true" : "false";

    // test string member-function empty
    cout << "\n\nTesting s3.empty():" << endl;

    if ( s3.empty() )
    {
        cout << "s3 is empty; assigning s1 to s3;" << endl;
        s3 = s1; // assign s1 to s3
        cout << "s3 is " "" "s3 << ""
    }
    else
    {
    }

    // test overloaded string concatenation operator
    cout << "\n\ns1 += s2 yields s1 = ";
    s1 += s2; // test overloaded concatenation
    cout << s1;

    // test overloaded string concatenation operator with C-style string
    cout << "\n\ns1 += \" to you\" yields" << endl;
    s1 += " to you";
    cout << "s1 = " "s1 << "\"\n";
```

Fig. 19.12 | Standard Library class string. (Part 1 of 3.)
// test string member function substr
cout << "The substring of s1 starting at location 0 for 14 characters, s1.substr(0, 14), is:\n" << s1.substr(0, 14) << endl;

// test substr "to-end-of-string" option
cout << "The substring of s1 starting at location 15, s1.substr(15), is:\n" << s1.substr(15) << endl;

// test copy constructor
string s4(s1);
cout << "\ns4 = " << s4 << endl;

// test overloaded assignment (=) operator with self-assignment
cout << "assigning s4 to s4" << endl;
s4 = s4;
cout << "s4 = " << s4 << endl;

// test using overloaded subscript operator to create lvalue
s1[0] = 'H';
s1[6] = 'B';
cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: " << s1 << endl;

// test subscript out of range with string member function "at"
cout << "Attempt to assign 'd' to s1.at(30) yields:" << endl;
s1.at(30) = 'd'; // ERROR: subscript out of range
}
} // end main

s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 > s1 yields false
s2 < s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday
s1 += " to you" yields
s1 = happy birthday to you

The substring of s1 starting at location 0 for 14 characters, s1.substr(0, 14), is:
happy birthday
The substring of s1 starting at location 15, s1.substr(15), is:

```
s4 = happy birthday to you
assigning s4 to s4
s4 = happy birthday to you
```

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1.at(30) yields:

This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information.

Fig. 19.12 | Standard Library class string. (Part 3 of 3.)

Lines 9–11 create three string objects—s1 is initialized with the literal "happy", s2 is initialized with the literal " birthday" and s3 uses the default string constructor to create an empty string. Lines 14–15 output these three objects, using cout and operator <<, which the string class designers overloaded to handle string objects. Then lines 16–22 show the results of comparing s2 to s1 by using class string’s overloaded equality and relational operators, which perform lexicographical comparisons using the numerical values of the characters (see Appendix B, ASCII Character Set) in each string.

Class string provides member function empty to determine whether a string is empty, which we demonstrate in line 27. Member function empty returns true if the string is empty; otherwise, it returns false.

Line 30 demonstrates class string’s overloaded assignment operator by assigning s1 to s3. Line 31 outputs s3 to demonstrate that the assignment worked correctly.

Line 36 demonstrates class string’s overloaded += operator for string concatenation. In this case, the contents of s2 are appended to s1. Then line 37 outputs the resulting string that is stored in s1. Line 41 demonstrates that a C-style string literal can be appended to a string object by using operator +=. Line 42 displays the result.

Class string provides member function substr (lines 47 and 52) to return a portion of a string as a string object. The call to substr in line 47 obtains a 14-character substring (specified by the second argument) of s1 starting at position 0 (specified by the first argument). The call to substr in line 52 obtains a substring starting from position 15 of s1. When the second argument is not specified, substr returns the remainder of the string on which it’s called.

Line 55 creates string object s4 and initializes it with a copy of s1. This results in a call to class string’s copy constructor. Line 60 uses class string’s overloaded = operator to demonstrate that it handles self-assignment properly.

Lines 64–65 use class string’s overloaded [] operator to create lvalues that enable new characters to replace existing characters in s1. Line 67 outputs the new value of s1. Class string’s overloaded [] operator does not perform any bounds checking. Therefore, you must ensure that operations using standard class string’s overloaded [] operator do not accidentally manipulate elements outside the bounds of the string. Class string does provide bounds checking in its member function at, which “throws an exception” if its argument is an invalid subscript. By default, this causes a C++ program to terminate and display a system-specific error message.³ If the subscript is valid, function at returns the
Chapter 19  Operator Overloading

character at the specified location as a modifiable *lvalue* or an unmodifiable *lvalue* (i.e., a const reference), depending on the context in which the call appears. Line 71 demonstrates a call to function at with an invalid subscript. The error message shown at the end of this program’s output was produced when running the program on Windows Vista.

### 19.15 *explicit* Constructors

In Sections 19.9–19.10, we discussed that any single-argument constructor can be used by the compiler to perform an implicit conversion—the type received by the constructor is converted to an object of the class in which the constructor is defined. The conversion is automatic and you need not use a cast operator. In some situations, implicit conversions are undesirable or error-prone. For example, our Array class in Fig. 19.6 defines a constructor that takes a single int argument. The intent of this constructor is to create an Array object containing the number of elements specified by the int argument. However, this constructor can be misused by the compiler to perform an implicit conversion.

**Common Programming Error 19.10**

Unfortunately, the compiler might use implicit conversions in cases that you do not expect, resulting in ambiguous expressions that generate compilation errors or result in execution-time logic errors.

Accidentally Using a Single-Argument Constructor as a Conversion Constructor

The program (Fig. 19.13) uses the Array class of Figs. 19.6–19.7 to demonstrate an improper implicit conversion.

```cpp
// Fig. 19.13: Fig19_13.cpp
// Driver for simple class Array.

#include <iostream>  // Standard I/O stream library
#include "Array.h"   // Declaration of the array class
using namespace std;

void outputArray( const Array & ); // prototype

int main()
{
    Array integers1( 7 ); // 7-element array
    outputArray( integers1 ); // output Array integers1
    outputArray( 3 ); // convert 3 to an Array and output Array's contents
}

// print Array contents
void outputArray( const Array &arrayToOutput )
{
    cout << "The Array received has " << arrayToOutput.getSize() << " elements. The contents are:

    " << arrayToOutput << endl;
}
```

*Fig. 19.13* | Single-argument constructors and implicit conversions. (Part 1 of 2.)

3. Again, Chapter 24 demonstrates how to “catch” and handle such exceptions.
Line 11 in `main` instantiates `Array` object `integers1` and calls the single argument constructor with the `int` value 7 to specify the number of elements in the `Array`. Recall from Fig. 19.7 that the `Array` constructor that receives an `int` argument initializes all the array elements to 0. Line 12 calls function `outputArray` (defined in lines 17–21), which receives as its argument a `const Array &` to an `Array`. The function outputs the number of elements in its `Array` argument and the contents of the `Array`. In this case, the size of the `Array` is 7, so seven 0s are output.

Line 13 calls function `outputArray` with the `int` value 3 as an argument. However, this program does not contain a function called `outputArray` that takes an `int` argument. So, the compiler determines whether class `Array` provides a conversion constructor that can convert an `int` into an `Array`. Since any constructor that receives a single argument is considered to be a conversion constructor, the compiler assumes the `Array` constructor that receives a single `int` is a conversion constructor and uses it to convert the argument 3 into a temporary `Array` object that contains three elements. Then, the compiler passes the temporary `Array` object to function `outputArray` to output the `Array`'s contents. Thus, even though we do not explicitly provide an `outputArray` function that receives an `int` argument, the compiler is able to compile line 13. The output shows the contents of the three-element `Array` containing 0s.

### Preventing Implicit Conversions with Single-Argument Constructors

C++ provides the keyword `explicit` to suppress implicit conversions via conversion constructors when such conversions should not be allowed. A constructor that is declared `explicit` cannot be used in an implicit conversion. Figure 19.14 declares an `explicit` constructor in class `Array`. The only modification to `Array.h` was the addition of the keyword `explicit` to the declaration of the single-argument constructor in line 14. No modifications are required to the source-code file containing class `Array`'s member-function definitions.

```cpp
// Fig. 19.14: Array.h
// Array class for storing arrays of integers.
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
using namespace std;

# define ARRAY_H

#include <iostream>
using namespace std;

Fig. 19.14 | Array class definition with explicit constructor. (Part 1 of 2.)
```
Chapter 19  Operator Overloading

Figure 19.15 presents a slightly modified version of the program in Fig. 19.13. When this program is compiled, the compiler produces an error message indicating that the integer value passed to outputArray in line 13 cannot be converted to a const Array &. The compiler error message (from Visual C++) is shown in the output window. Line 14 demonstrates how the explicit constructor can be used to create a temporary Array of 3 elements and pass it to function outputArray.

```cpp
class Array
{
    friend ostream &operator<<(ostream &, const Array &);
    friend istream &operator>>(istream &, Array &);

public:
    explicit Array(int = 10); // default constructor
    Array(const Array &); // copy constructor
    ~Array(); // destructor
    int getSize() const; // return size

    const Array &operator=(const Array &); // assignment operator
    bool operator==(const Array &) const; // equality operator

    // inequality operator; returns opposite of == operator
    bool operator!=(const Array &right) const
    {
        return !(*(this) == right); // invokes Array::operator==
    } // end function operator!=

    // subscript operator for non-const objects returns lvalue
    int &operator[](int);

    // subscript operator for const objects returns rvalue
    const int &operator[](int) const;

private:
    int size; // pointer-based array size
    int *ptr; // pointer to first element of pointer-based array
}; // end class Array
```

Figure 19.14  Array class definition with explicit constructor. (Part 2 of 2.)

Figure 19.15 presents a slightly modified version of the program in Fig. 19.13. When this program is compiled, the compiler produces an error message indicating that the integer value passed to outputArray in line 13 cannot be converted to a const Array &. The compiler error message (from Visual C++) is shown in the output window. Line 14 demonstrates how the explicit constructor can be used to create a temporary Array of 3 elements and pass it to function outputArray.

```cpp
// Fig. 19.15: Fig19_15.cpp
// Driver for simple class Array.
#include <iostream>
#include "Array.h"
using namespace std;

void outputArray(const Array &); // prototype
```

Figure 19.15  Demonstrating an explicit constructor. (Part 1 of 2.)
Recall that two of the fundamental principles of good software engineering are separating interface from implementation and hiding implementation details. We strive to achieve these goals by defining a class in a header file and implementing its member functions in a separate implementation file. As we pointed out in Chapter 17, however, header files do contain a portion of a class’s implementation and hints about others. For example, a class’s private members are listed in the class definition in a header file, so these members are visible to clients, even though the clients may not access the private members. Revealing a class’s private data in this manner potentially exposes proprietary information to clients of the class. We now introduce the notion of a proxy class that allows you to hide even the private data of a class from clients of the class. Providing clients of your class with a proxy class that knows only the public interface to your class enables the clients to use your class’s services without giving the clients access to your class’s implementation details.

Implementing a proxy class requires several steps, which we demonstrate in Figs. 19.16–19.19. First, we create the class definition for the class that contains the proprietary implementation we’d like to hide. Our example class, called Implementation, is

```cpp
int main()
{
  Array integers1( 7 ); // 7-element array
  outputArray( integers1 ); // output Array integers1
  outputArray( 3 ); // convert 3 to an Array and output Array’s contents
  outputArray( Array( 3 ) ); // explicit single-argument constructor call
}
```

// print array contents
void outputArray( const Array &arrayToOutput )
{
  cout << "The Array received has " << arrayToOutput.getSize()
       << " elements. The contents are:\n" << arrayToOutput << endl;
}
```

Fig. 19.15 | Demonstrating an explicit constructor. (Part 2 of 2.)

Common Programming Error 19.11

Attempting to invoke an explicit constructor for an implicit conversion is a compilation error.

Error-Prevention Tip 19.3

Use the explicit keyword on single-argument constructors that should not be used by the compiler to perform implicit conversions.
Chapter 19  Operator Overloading

shown in Fig. 19.16. The proxy class Interface is shown in Figs. 19.17–19.18. The test program and sample output are shown in Fig. 19.19.

Class Implementation (Fig. 19.16) provides a single private data member called value (the data we'd like to hide from the client), a constructor to initialize value and functions setValue and getValue.

---

```c++
// Fig. 19.16: Implementation.h
// Implementation class definition.

class Implementation {
public:
    // constructor
    Implementation(int v) : value(v) // initialize value with v
        {
        // empty body
    } // end constructor Implementation

    // set value to v
    void setValue(int v) {
        value = v; // should validate v
    } // end function setValue

    // return value
    int getValue() const {
        return value;
    } // end function getValue

private:
    int value; // data that we would like to hide from the client
}; // end class Implementation
```

---

Fig. 19.16  |  Implementation class definition.

We define a proxy class called Interface (Fig. 19.17) with an identical public interface (except for the constructor and destructor names) to that of class Implementation. The proxy class’s only private member is a pointer to an Implementation object. Using a pointer in this manner allows us to hide class Implementation’s implementation details from the client. Notice that the only mentions in class Interface of the proprietary Implementation class are in the pointer declaration (line 17) and in line 6, a forward class declaration. When a class definition uses only a pointer or reference to an object of another class (as in this case), the class header file for that other class (which would ordinarily reveal the private data of that class) is not required to be included with #include. This is because the compiler doesn’t need to reserve space for an object of the class. The compiler does need to reserve space for the pointer or reference. The sizes of pointers and references are characteristics of the hardware platform on which the compiler runs, so the compiler already knows those sizes. You can simply declare that other class as a data type with a forward class declaration (line 6) before the type is used in the file.
19.16 Proxy Classes

The member-function implementation file for proxy class Interface (Fig. 19.18) is the only file that includes the header file Implementation.h (line 5) containing class Implementation. The file Interface.cpp (Fig. 19.18) is provided to the client as a pre-compiled object code file along with the header file Interface.h that includes the function prototypes of the services provided by the proxy class. Because file Interface.cpp is made available to the client only as object code, the client is not able to see the interactions between the proxy class and the proprietary class (lines 9, 17, 23 and 29). The proxy class imposes an extra “layer” of function calls as the “price to pay” for hiding the private data of class Implementation. Given the speed of today’s computers and the fact that many compilers can inline simple function calls automatically, the effect of these extra function calls on performance is often negligible.

Fig. 19.17 | Proxy class Interface definition.

```cpp
// Fig. 19.17: Interface.h
// Proxy class Interface definition.
// Client sees this source code, but the source code does not reveal
// the data layout of class Implementation.

// forward class declaration required by line 17
class Implementation;

class Interface
{
    public:
        Interface( int ); // constructor
        void setValue( int ); // same public interface as
        int getValue() const; // class Implementation has
        ~Interface(); // destructor

    private:
        // requires previous forward declaration (line 6)
        Implementation *ptr;
}; // end class Interface
```

Fig. 19.17 | Proxy class Interface definition.

Fig. 19.18 | Interface class member-function definitions. (Part 1 of 2.)

```cpp
// Fig. 19.18: Interface.cpp
// Implementation of class Interface--client receives this file only
// as precompiled object code, keeping the implementation hidden.
#include "Interface.h" // Interface class definition
#include "Implementation.h" // Implementation class definition

// constructor
Interface::Interface( int v ) : ptr( new Implementation( v ) ) // initialize ptr to point to
    // a new Implementation object
{
    // empty body
} // end Interface constructor

// call Implementation's setValue function
void Interface::setValue( int v )
{
```
Chapter 19  Operator Overloading

Figure 19.19 tests class Interface. Notice that only the header file for Interface is included in the client code (line 4)—there is no mention of the existence of a separate class called Implementation. Thus, the client never sees the private data of class Implementation, nor can the client code become dependent on the Implementation code.

Software Engineering Observation 19.9

A proxy class insulates client code from implementation changes.

Fig. 19.19  |  Implementing a proxy class.
19.17 Wrap-Up

In this chapter, you learned how to build more robust classes by defining overloaded operators that enable you to use operators with objects of your classes. We presented basic operator overloading concepts, as well as several restrictions that the C++ standard places on overloaded operators. You learned reasons for implementing overloaded operators as member functions or as global functions. We discussed the differences between overloading unary and binary operators as member functions and global functions. With global functions, we showed how to input and output objects of our classes using the overloaded stream extraction and stream insertion operators, respectively. We introduced the concept of dynamic memory management. You learned that you can create and destroy objects dynamically with the `new` and `delete` operators, respectively. We showed a special syntax that is required to differentiate between the prefix and postfix versions of the increment (`++`) operator. We also demonstrated standard C++ class `string`, which makes extensive use of overloaded operators to create a robust, reusable class that can replace C-style, pointer-based strings. You learned how to use keyword `explicit` to prevent the compiler from using a single-argument constructor to perform implicit conversions. Finally, we showed how to create a proxy class to hide the implementation details of a class from the class’s clients. In the next chapter, we continue our discussion of classes by introducing a form of software reuse called inheritance. We’ll see that when classes share common attributes and behaviors, it’s possible to define those attributes and behaviors in a common “base” class and “inherit” those capabilities into new class definitions, enabling you to create the new classes with a minimal amount of code.

Summary

Section 19.1 Introduction

- C++ enables you to overload most operators to be sensitive to the context in which they’re used—the compiler generates the appropriate code based on the context (in particular, the types of the operands).
- Many of C++’s operators can be overloaded to work with user-defined types.
- One example of an overloaded operator built into C++ is operator `<<`, which is used both as the stream insertion operator and as the bitwise left-shift operator. Similarly, `>>` is also overloaded; it’s used both as the stream extraction operator and as the bitwise right-shift operator. Both of these operators are overloaded in the C++ Standard Library.
- The C++ language itself overloads `+` and `-`. These operators perform differently, depending on their context in integer arithmetic, floating-point arithmetic and pointer arithmetic.
- The jobs performed by overloaded operators can also be performed by function calls, but operator notation is often clearer and more familiar to programmers.

Section 19.2 Fundamentals of Operator Overloading

- An operator is overloaded by writing a non-static member-function definition or global function definition in which the function name is the keyword `operator` followed by the symbol for the operator being overloaded.
- When operators are overloaded as member functions, they must be non-static, because they must be called on an object of the class and operate on that object.
• To use an operator on class objects, that operator must be overloaded, with three exceptions—the assignment operator (\(=\)), the address operator (\&) and the comma operator (\(,\)).

Section 19.3 Restrictions on Operator Overloading
• You cannot change the precedence and associativity of an operator by overloading.
• You cannot change the “arity” of an operator (i.e., the number of operands an operator takes).
• You cannot create new operators—only existing operators can be overloaded.
• You cannot change the meaning of how an operator works on objects of fundamental types.
• Overloading an assignment operator and an addition operator for a class does not imply that the += operator is also overloaded. Such behavior can be achieved only by explicitly overloading operator += for that class.

Section 19.4 Operator Functions as Class Members vs. Global Function
• Operator functions can be member functions or global functions—global functions are often made friends for performance reasons. Member functions use the this pointer implicitly to obtain one of their class object arguments (the left operand for binary operators). Arguments for both operands of a binary operator must be explicitly listed in a global function call.
• Overloaded \((), [], \rightarrow\) and assignment operators must be declared as class members. For the other operators, the operator overloading functions can be class members or global functions.
• When an operator function is implemented as a member function, the leftmost (or only) operand must be an object (or a reference to an object) of the operator’s class.
• If the left operand must be an object of a different class or a fundamental type, this operator function must be implemented as a global function.
• A global operator function can be made a friend of a class if that function must access private or protected members of that class directly.

Section 19.5 Overloading Stream Insertion and Stream Extraction Operators
• The overloaded stream insertion operator (\(<<\)) is used in an expression in which the left operand has type ostream \&. For this reason, it must be overloaded as a global function. To be a member function, operator \(<\) would have to be a member of the ostream class, but this is not possible, since we’re not allowed to modify C++ Standard Library classes. Similarly, the overloaded stream extraction operator (\(>>\)) must be a global function.
• Another reason to choose a global function to overload an operator is to enable the operator to be commutative.
• When used with cin, setw restricts the number of characters read to the number of characters specified by its argument.
• istream member function ignore discards the specified number of characters in the input stream (one character by default).
• Overloaded input and output operators are declared as friends if they need to access non-public class members directly for performance reasons.

Section 19.6 Overloading Unary Operators
• A unary operator for a class can be overloaded as a non-static member function with no arguments or as a global function with one argument; that argument must be either an object of the class or a reference to an object of the class.
• Member functions that implement overloaded operators must be non-static so that they can access the non-static data in each object of the class.
Section 19.7 Overloading Binary Operators
• A binary operator can be overloaded as a non-static member function with one argument or as a global function with two arguments (one of those arguments must be either a class object or a reference to a class object).

Section 19.8 Dynamic Memory Management
• Dynamic memory management enables you to control the allocation and deallocation of memory in a program for any built-in or user-defined type.
• The free store (sometimes called the heap) is a region of memory assigned to each program for storing objects dynamically allocated at execution time.
• The new operator allocates storage of the proper size for an object, runs the object’s constructor and returns a pointer of the correct type. The new operator can be used to dynamically allocate any fundamental type (such as int or double) or class type. If new is unable to find space in memory for the object, it indicates that an error occurred by “throwing” an “exception.” This usually causes the program to terminate immediately, unless the exception is handled.
• To destroy a dynamically allocated object and free its space, use the delete operator.
• An array of objects can be allocated dynamically with new as in

```cpp
int *ptr = new int[100];
```

which allocates an array of 100 integers and assigns the starting location of the array to ptr. The preceding array of integers is deleted with the statement

```cpp
delete [] ptr;
```

Section 19.9 Case Study: Array Class
• A copy constructor initializes a new object of a class by copying the members of an existing one. Classes that contain dynamically allocated memory, typically provide a copy constructor, a destructor and an overloaded assignment operator.
• The implementation of member function operator= should test for self-assignment, in which an object is being assigned to itself.
• The compiler calls the const version of operator[] when the subscript operator is used on a const object and calls the non-const version of the operator when it’s used on a non-const object.
• The array subscript operator ([] ) can be used to select elements from other types of containers. Also, with overloading, the index values no longer need to be integers.

Section 19.10 Converting between Types
• The compiler cannot know in advance how to convert among user-defined types, and between user-defined types and fundamental types, so you must specify how to do this. Such conversions can be performed with conversion constructors—single-argument constructors that turn objects of other types (including fundamental types) into objects of a particular class.
• Any single-argument constructor can be thought of as a conversion constructor.
• A conversion operator can be used to convert an object of one class into an object of another class or into an object of a fundamental type. Such a conversion operator must be a non-static member function. Overloaded cast-operator functions can be defined for converting objects of user-defined types into fundamental types or into objects of other user-defined types.
• An overloaded cast operator function does not specify a return type—the return type is the type to which the object is being converted.
• When necessary, the compiler can call cast operators and conversion constructors implicitly to create temporary objects.
Section 19.11 Building a String Class
- Overloading the function call operator () is powerful, because functions can take an arbitrary number of parameters.

Section 19.12 Overloading ++ and --
- The prefix and postfix increment and decrement operator can all be overloaded.
- To overload the pre- and post-increment operators, each overloaded operator function must have a distinct signature. The prefix versions are overloaded like any other unary operator. The postfix increment operator’s unique signature is accomplished by providing a second argument, which must be of type int. This argument is not supplied in the client code. It’s used implicitly by the compiler to distinguish between the prefix and postfix versions of the increment operator. The same syntax is used to differentiate between the prefix and postfix decrement operator functions.

Section 19.14 Standard Library Class string
- Standard class string is defined in header <string> and belongs to namespace std.
- Class string provides many overloaded operators, including equality, relational, assignment, addition assignment (for concatenation) and subscript operators.
- Class string provides member function empty, which returns true if the string is empty; otherwise, it returns false.
- Standard class string member function substr obtains a substring of a length specified by the second argument, starting at the position specified by the first argument. When the second argument is not specified, substr returns the remainder of the string on which it’s called.
- Class string’s overloaded [] operator does not perform any bounds checking. Therefore, you must ensure that operations using standard class string’s overloaded [] operator do not accidentally manipulate elements outside the bounds of the string.
- Standard class string provides bounds checking with member function at, which “throws an exception” if its argument is an invalid subscript. By default, this causes the program to terminate. If the subscript is valid, function at returns a reference or a const reference to the character at the specified location depending on the context.

Section 19.15 explicit Constructors
- C++ provides the keyword explicit to suppress implicit conversions via conversion constructors when such conversions should not be allowed. A constructor that is declared explicit cannot be used in an implicit conversion.

Section 19.16 Proxy Classes
- Providing clients of your class with a proxy class that knows only the public interface to your class enables the clients to use your class’s services without giving the clients access to your class’s implementation details, such as its private data.
- When a class definition uses only a pointer or reference to an object of another class, the class header file for that other class (which would ordinarily reveal the private data of that class) is not required to be included with #include. You can simply declare that other class as a data type with a forward class declaration before the type is used in the file.
- The implementation file containing the member functions for a proxy class is the only file that includes the header file for the class whose private data we’d like to hide.
- The implementation file containing the member functions for the proxy class is provided to the client as a precompiled object code file along with the header file that includes the function prototypes of the services provided by the proxy class.
Terminology

at member function of class string 707
allocate memory 682
cast operator function 697
conversion constructor 696
conversion operator 696
copy constructor 691
dangling pointer 694
deallocate memory 682
delete operator 682
delete[] operator 682
dynamic memory management 682
empty member function of string 707
explicit constructor 709
forward class declaration 712
free store 682
function call operator () 698
heap 682
initializer 683
memory leak 683
new operator 682
new[] operator 683
operator overloading 673
proxy class 711
self-assignment 694
single-argument constructor 696
substr member function of class string 707

Self-Review Exercises

19.1 Fill in the blanks in each of the following:
a) Suppose a and b are integer variables and we form the sum a + b. Now suppose c and
d are floating-point variables and we form the sum c + d. The two + operators here are
clearly being used for different purposes. This is an example of ________.
b) Keyword ________ introduces an overloaded-operator function definition.
c) To use operators on class objects, they must be overloaded, with the exception of opera-
tors ________, ________, and ________.
d) The ________, ________, and ________ of an operator cannot be changed by overload-
ing the operator.
e) The operators that cannot be overloaded are ________, ________, ________, and
________.
f) The ________ operator reclaims memory previously allocated by new.
g) The ________ operator dynamically allocates memory for an object of a specified type
and returns a(n) ________ to that type.

19.2 Explain the multiple meanings of the operators << and >>.

19.3 In what context might the name operator/ be used?

19.4 (True/False) Only existing operators can be overloaded.

19.5 How does the precedence of an overloaded operator compare with the precedence of the
original operator?

Answers to Self-Review Exercises

19.1 a) operator overloading. b) operator. c) assignment (=), address (&), comma (,).
d) precedence, associativity, “arity.” e) .?, .*, and .:: f) delete. g) new, pointer.

19.2 Operator >> is both the right-shift operator and the stream extraction operator, depending
on its context. Operator << is both the left-shift operator and the stream insertion operator, depending
on its context.

19.3 For operator overloading: It would be the name of a function that would provide an over-
loaded version of the / operator for a specific class.

19.4 True.

19.5 The precedence is identical.
Exercises

19.6 Compare and contrast dynamic memory allocation and deallocation operators new, new [], delete and delete [].

19.7 (Overloading the Parentheses Operator) One nice example of overloading the function call operator () is to allow another form of double-array subscripting popular in some programming languages. Instead of saying

```chessBoard[ row ][ column ]```

for an array of objects, overload the function call operator to allow the alternate form

```chessBoard( row, column )```

Create a class DoubleSubscriptedArray that has similar features to class Array in Figs. 19.6–19.7. At construction time, the class should be able to create an array of any number of rows and any number of columns. The class should supply operator() to perform double-subscripting operations. For example, in a 3-by-5 DoubleSubscriptedArray called a, the user could write `a(1, 3)` to access the element at row 1 and column 3. Remember that operator() can receive any number of arguments. The underlying representation of the double-subscripted array should be a single-subscripted array of integers with `rows * columns` number of elements. Function operator() should perform the proper pointer arithmetic to access each element of the array. There should be two versions of operator()—one that returns `int &` (so that an element of a DoubleSubscriptedArray can be used as an `lvalue`) and one that returns `const int &`. The class should also provide the following operators: `==, !=, =, <<=` (for outputting the array in row and column format) and `>>` (for inputting the entire array contents).

19.8 (Complex Class) Consider class Complex shown in Figs. 19.20–19.22. The class enables operations on so-called complex numbers. These are numbers of the form `realPart + imaginaryPart * i`, where `i` has the value `sqrt(-1)

a) Modify the class to enable input and output of complex numbers via overloaded `>>` and `<<` operators, respectively (you should remove the `print` function from the class).

b) Overload the multiplication operator to enable multiplication of two complex numbers as in algebra.

c) Overload the `==` and `!=` operators to allow comparisons of complex numbers.
Exercises

Fig. 19.20  |  Complex class definition. (Part 2 of 2.)

Fig. 19.21  |  Complex class member-function definitions.

Fig. 19.22  |  Complex numbers. (Part 1 of 2.)
Chapter 19  Operator Overloading

19.9 (HugeInt Class) A machine with 32-bit integers can represent integers in the range of approximately \(-2\) billion to \(+2\) billion. This fixed-size restriction is rarely troublesome, but there are applications in which we’d like to be able to use a much wider range of integers. This is what C++ was built to do, namely, create powerful new data types. Consider class HugeInt of Figs. 19.23–19.25. Study the class carefully, then answer the following:

a) Describe precisely how it operates.

b) What restrictions does the class have?

c) Overload the * multiplication operator.

d) Overload the / division operator.

e) Overload all the relational and equality operators.

[Note: We do not show an assignment operator or copy constructor for class HugeInteger, because the assignment operator and copy constructor provided by the compiler are capable of copying the entire array data member properly.]
using namespace std;

class HugeInt
{
friend ostream &operator<< ( ostream &, const HugeInt &);

public:
static const int digits = 30; // maximum digits in a HugeInt
HugeInt ( long = 0 ); // conversion/default constructor
HugeInt ( const string & ); // conversion constructor

// addition operator; HugeInt + HugeInt
HugeInt operator + ( const HugeInt & ) const;

// addition operator; HugeInt + int
HugeInt operator + ( int ) const;

// addition operator;
// HugeInt + string that represents large integer value
HugeInt operator + ( const string & ) const;

private:
    short integer[ digits ];
}; // end class HugeInt

/* Fig. 19.23 | HugeInt class definition. (Part 2 of 2.) */

// Fig. 19.24: Hugeint.cpp
// HugeInt member-function and friend-function definitions.
#include <cctype> // isdigit function prototype
#include "Hugeint.h" // HugeInt class definition
using namespace std;

// default constructor; conversion constructor that converts
// a long integer into a HugeInt object
HugeInt::HugeInt( long value )
{
    // initialize array to zero
    for ( int i = 0; i < digits; i++ )
        integer[ i ] = 0;

    // place digits of argument into array
    for ( int j = digits - 1; value != 0 && j >= 0; j-- )
    {
        integer[ j ] = value % 10;
        value /= 10;
    } // end for

    // end HugeInt default/conversion constructor

    // conversion constructor that converts a character string
    // representing a large integer into a HugeInt object
    HugeInt::HugeInt( const string &number )
    {
        // initialize array to zero
        for ( int i = 0; i < digits; i++ )
            integer[ i ] = 0;

        // place digits of argument into array
        for ( int j = digits - 1; value != 0 && j >= 0; j-- )
        {
            integer[ j ] = value % 10;
            value /= 10;
        } // end for

    } // end HugeInt conversion constructor

/* Fig. 19.24 | HugeInt class member-function and friend-function definitions. (Part 1 of 3.) */
// place digits of argument into array
int length = number.size;

for ( int j = digits - length, k = 0; j < digits; j++, k++ )
    if ( isdigit( number[ k ] ) ) // ensure that character is a digit
        integer[ j ] = number[ k ] - '0';

} // end HugeInt conversion constructor

// addition operator; HugeInt + HugeInt
HugeInt HugeInt::operator+( const HugeInt &op2 ) const
{
    HugeInt temp; // temporary result
    int carry = 0;

    for ( int i = digits - 1; i >= 0; i-- )
    {
        temp.integer[ i ] = integer[ i ] + op2.integer[ i ] + carry;

        // determine whether to carry a 1
        if ( temp.integer[ i ] > 9 )
            { temp.integer[ i ] %= 10; // reduce to 0-9
                carry = 1;
            } // end if
        else // no carry
            carry = 0;
    } // end for

    return temp; // return copy of temporary object
} // end function operator+

// addition operator; HugeInt + int
HugeInt HugeInt::operator+( int op2 ) const
{
    // convert op2 to a HugeInt, then invoke
    // operator+ for two HugeInt objects
    return *this + HugeInt( op2 );
} // end function operator+

// addition operator;
// HugeInt + string that represents large integer value
HugeInt HugeInt::operator+( const string &op2 ) const
{
    // convert op2 to a HugeInt, then invoke
    // operator+ for two HugeInt objects
    return *this + HugeInt( op2 );
} // end operator+

// overloaded output operator
ostream& operator<<( ostream &output, const HugeInt &num )
{
    int i;

    for ( i = 0; num.integer[ i ] == 0 ) && ( i <= HugeInt::digits ); i++ )
        ; // skip leading zeros

    if ( i == HugeInt::digits )
        output << 0;

Fig. 19.24 | HugeInt class member-function and friend-function definitions. (Part 2 of 3.)
Exercise 19.10 \textit{(RationalNumber Class)} Create a class \texttt{RationalNumber} (fractions) with the following capabilities:

a) Create a constructor that prevents a 0 denominator in a fraction, reduces or simplifies fractions that are not in reduced form and avoids negative denominators.
b) Overload the addition, subtraction, multiplication and division operators for this class.
c) Overload the relational and equality operators for this class.

19.11 (Polynomial Class) Develop class Polynomial. The internal representation of a Polynomial is an array of terms. Each term contains a coefficient and an exponent, e.g., the term
\[ 2x^4 \]
has the coefficient 2 and the exponent 4. Develop a complete class containing proper constructor and destructor functions as well as set and get functions. The class should also provide the following overloaded operator capabilities:

a) Overload the addition operator (+) to add two Polynomials.
b) Overload the subtraction operator (−) to subtract two Polynomials.
c) Overload the assignment operator to assign one Polynomial to another.
d) Overload the multiplication operator (*) to multiply two Polynomials.
e) Overload the addition assignment operator (+=), subtraction assignment operator (−=), and multiplication assignment operator (*=).
Say not you know another entirely, till you have divided an inheritance with him.
—Johann Kasper Lavater

This method is to define as the number of a class the class of all classes similar to the given class.
—Bertrand Russell

Good as it is to inherit a library, it is better to collect one.
—Augustine Birrell

Save base authority from others’ books.
—William Shakespeare

Objectives
In this chapter you’ll learn:

■ To create classes by inheriting from existing classes.

■ The notions of base classes and derived classes and the relationships between them.

■ The protected member access specifier.

■ The use of constructors and destructors in inheritance hierarchies.

■ The order in which constructors and destructors are called in inheritance hierarchies.

■ The differences between public, protected and private inheritance.

■ To use inheritance to customize existing software.
Chapter 20  Object-Oriented Programming: Inheritance

20.1 Introduction

This chapter continues our discussion of object-oriented programming (OOP) by introducing another of its key features—inheritance. Inheritance is a form of software reuse in which you create a class that absorbs an existing class’s data and behaviors and enhances them with new capabilities. Software reusability saves time during program development. It also encourages the reuse of proven, debugged, high-quality software, which increases the likelihood that a system will be implemented effectively.

When creating a class, instead of writing completely new data members and member functions, you can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class. (Other programming languages, such as Java, refer to the base class as the superclass and the derived class as the subclass.) A derived class represents a more specialized group of objects. Typically, a derived class contains behaviors inherited from its base class plus additional behaviors. As we’ll see, a derived class can also customize behaviors inherited from the base class. A direct base class is the base class from which a derived class explicitly inherits. An indirect base class is inherited from two or more levels up in the class hierarchy. In the case of single inheritance, a class is derived from one base class. C++ also supports multiple inheritance, in which a derived class inherits from multiple (possibly unrelated) base classes. Single inheritance is straightforward—we show several examples that should enable you to become proficient quickly. Multiple inheritance can be complex and error prone.

C++ offers public, protected and private inheritance. In this chapter, we concentrate on public inheritance and briefly explain the other two. We do not discuss private inheritance in detail. The third form, protected inheritance, is rarely used. With public inheritance, every object of a derived class is also an object of that derived class’s base class. However, base-class objects are not objects of their derived classes. For example, if we have vehicle as a base class and car as a derived class, then all cars are vehicles, but not all vehicles are cars. As we continue our study of object-oriented programming in this chapter and...
Chapter 21, we take advantage of this relationship to perform some interesting manipulations.

Experience in building software systems indicates that significant amounts of code deal with closely related special cases. When you’re preoccupied with special cases, the details can obscure the big picture. With object-oriented programming, you focus on the commonalities among objects in the system rather than on the special cases.

We distinguish between the *is-a* relationship and the *has-a* relationship. The *is-a* relationship represents inheritance. In an *is-a* relationship, an object of a derived class also can be treated as an object of its base class—for example, a car *is* a vehicle, so any attributes and behaviors of a vehicle are also attributes and behaviors of a car. By contrast, the *has-a* relationship represents composition. (Composition was discussed in Chapter 18.) In a *has-a* relationship, an object contains one or more objects of other classes as members. For example, a car includes many components—it *has* a steering wheel, *has* a brake pedal, *has* a transmission and *has* many other components.

Derived-class member functions might require access to base-class data members and member functions. A derived class can access the non-private members of its base class. Base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class. A derived class can change the values of private base-class members, but only through non-private member functions provided in the base class and inherited into the derived class.

---

**Software Engineering Observation 20.1**

Member functions of a derived class cannot directly access private members of the base class.

**Software Engineering Observation 20.2**

If a derived class could access its base class’s private members, classes that inherit from that derived class could access that data as well. This would propagate access to what should be private data, and the benefits of information hiding would be lost.

---

One problem with inheritance is that a derived class can inherit data members and member functions it does not need or should not have. It’s the class designer’s responsibility to ensure that the capabilities provided by a class are appropriate for future derived classes. Even when a base-class member function is appropriate for a derived class, the derived class often requires that the member function behave in a manner specific to the derived class. In such cases, the base-class member function can be redefined in the derived class with an appropriate implementation.

20.2 Base Classes and Derived Classes

Often, an object of one class *is an* object of another class, as well. For example, in geometry, a rectangle *is a* quadrilateral (as are squares, parallelograms and trapezoids). Thus, in C++, class `Rectangle` can be said to *inherit* from class `Quadrilateral`. In this context, class `Quadrilateral` is a base class, and class `Rectangle` is a derived class. A rectangle *is a* specific type of quadrilateral, but it’s incorrect to claim that a quadrilateral *is a* rectangle—the quadrilateral could be a parallelogram or some other shape. Figure 20.1 lists several simple examples of base classes and derived classes.
Because every derived-class object is an object of its base class, and one base class can have many derived classes, the set of objects represented by a base class typically is larger than the set of objects represented by any of its derived classes. For example, the base class Vehicle represents all vehicles, including cars, trucks, boats, airplanes, bicycles and so on. By contrast, derived class Car represents a smaller, more specific subset of all vehicles.

Inheritance relationships form treelike hierarchical structures. A base class exists in a hierarchical relationship with its derived classes. Although classes can exist independently, once they’re employed in inheritance relationships, they become affiliated with other classes. A class becomes either a base class—supplying members to other classes, a derived class—inheriting its members from other classes, or both.

Let’s develop a simple inheritance hierarchy with five levels (represented by the UML class diagram in Fig. 20.2). A university community has thousands of members.

**Fig. 20.2 | Inheritance hierarchy for university CommunityMembers.**

These members consist of employees, students and alumni. Employees are either faculty members or staff members. Faculty members are either administrators (such as deans and department chairpersons) or teachers. Some administrators, however, also teach
20.2 Base Classes and Derived Classes

classes. Note that we’ve used multiple inheritance to form class AdministratorTeacher. Also, this inheritance hierarchy could contain many other classes. For example, students can be graduate or undergraduate students. Undergraduate students can be freshmen, sophomores, juniors and seniors.

Each arrow in the hierarchy (Fig. 20.2) represents an is-a relationship. For example, as we follow the arrows in this class hierarchy, we can state “an Employee is a CommunityMember” and “a Teacher is a Faculty member.” CommunityMember is the direct base class of Employee, Student and Alumnus. In addition, CommunityMember is an indirect base class of all the other classes in the diagram. Starting from the bottom of the diagram, you can follow the arrows and apply the is-a relationship to the topmost base class. For example, an AdministratorTeacher is an Administrator, is a Faculty member, is an Employee and is a CommunityMember.

Now consider the Shape inheritance hierarchy in Fig. 20.3. This hierarchy begins with base class Shape. Classes TwoDimensionalShape and ThreeDimensionalShape derive from base class Shape—Shapes are either TwoDimensionalShapes or ThreeDimensionalShapes. The third level of this hierarchy contains some more specific types of TwoDimensionalShapes and ThreeDimensionalShapes. As in Fig. 20.2, we can follow the arrows from the bottom of the diagram to the topmost base class in this class hierarchy to identify several is-a relationships. For instance, a Triangle is a TwoDimensionalShape and is a Shape, while a Sphere is a ThreeDimensionalShape and is a Shape. This hierarchy could contain many other classes, such as Rectangles, Ellipses and Trapezoids, which are all TwoDimensionalShapes.

To specify that class TwoDimensionalShape (Fig. 20.3) is derived from (or inherits from) class Shape, class TwoDimensionalShape’s definition could begin as follows:

```cpp
class TwoDimensionalShape : public Shape
```

This is an example of public inheritance, the most commonly used form. We also will discuss private inheritance and protected inheritance (Section 20.6). With all forms of inheritance, private members of a base class are not accessible directly from that class’s derived classes, but these private base-class members are still inherited (i.e., they’re still considered parts of the derived classes). With public inheritance, all other base-class members retain their original member access when they become members of the derived class (e.g., public members of the base class become public members of the derived class, and, as we’ll soon see, protected members of the base class become protected members of the
derived class). Through these inherited base-class members, the derived class can manipulate private members of the base class (if these inherited members provide such functionality in the base class). Note that friend functions are not inherited.

Inheritance is not appropriate for every class relationship. In Chapter 18, we discussed the has-a relationship, in which classes have members that are objects of other classes. Such relationships create classes by composition of existing classes. For example, given the classes Employee, BirthDate and TelephoneNumber, it’s improper to say that an Employee is a BirthDate or that an Employee is a TelephoneNumber. However, it’s appropriate to say that an Employee has a BirthDate and that an Employee has a TelephoneNumber.

It’s possible to treat base-class objects and derived-class objects similarly; their commonalities are expressed in the members of the base class. Objects of all classes derived from a common base class can be treated as objects of that base class (i.e., such objects have an is-a relationship with the base class). In Chapter 21, we consider many examples that take advantage of this relationship.

20.3 protected Members

Chapter 16 introduced access specifiers public and private. A base class’s public members are accessible within its body and anywhere that the program has a handle (i.e., a name, reference or pointer) to an object of that class or one of its derived classes. A base class’s private members are accessible only within its body and to the friends of that base class. In this section, we introduce the access specifier protected.

Using protected access offers an intermediate level of protection between public and private access. A base class’s protected members can be accessed within the body of that base class, by members and friends of that base class, and by members and friends of any classes derived from that base class.

Derived-class member functions can refer to public and protected members of the base class simply by using the member names. When a derived-class member function redefines a base-class member function, the base-class member can be accessed from the derived class by preceding the base-class member name with the base-class name and the binary scope resolution operator (::). We discuss accessing redefined members of the base class in Section 20.4.5 and using protected data in Section 20.4.4.

20.4 Relationship between Base Classes and Derived Classes

In this section, we use an inheritance hierarchy containing types of employees in a company’s payroll application to discuss the relationship between a base class and a derived class. Commission employees (who will be represented as objects of a base class) are paid a percentage of their sales, while base-salaried commission employees (who will be represented as objects of a derived class) receive a base salary plus a percentage of their sales. We divide our discussion of the relationship between commission employees and base-salaried commission employees into a carefully paced series of five examples:

1. In the first example, we create class CommissionEmployee, which contains as private data members a first name, last name, social security number, commission rate (percentage) and gross (i.e., total) sales amount.
2. The second example defines class `BasePlusCommissionEmployee`, which contains as private data members a first name, last name, social security number, commission rate, gross sales amount and base salary. We create the latter class by writing every line of code the class requires—we'll soon see that it's much more efficient to create this class simply by inheriting from class `CommissionEmployee`.

3. The third example defines a new version of class `BasePlusCommissionEmployee` class that inherits directly from class `CommissionEmployee` (i.e., a `BasePlusCommissionEmployee` is a `CommissionEmployee` who also has a base salary) and attempts to access class `CommissionEmployee`'s private members—this results in compilation errors, because the derived class does not have access to the base class's private data.

4. The fourth example shows that if `CommissionEmployee`'s data is declared as protected, a new version of class `BasePlusCommissionEmployee` that inherits from class `CommissionEmployee` can access that data directly. For this purpose, we define a new version of class `CommissionEmployee` with protected data. Both the inherited and noninherited `BasePlusCommissionEmployee` classes contain identical functionality, but we show how the version of `BasePlusCommissionEmployee` that inherits from class `CommissionEmployee` is easier to create and manage.

5. After we discuss the convenience of using protected data, we create the fifth example, which sets the `CommissionEmployee` data members back to private to enforce good software engineering. This example demonstrates that derived class `BasePlusCommissionEmployee` can use base class `CommissionEmployee`'s public member functions to manipulate `CommissionEmployee`'s private data.

### 20.4.1 Creating and Using a `CommissionEmployee` Class

Let's examine `CommissionEmployee`'s class definition (Figs. 20.4–20.5). The `CommissionEmployee` header file (Fig. 20.4) specifies class `CommissionEmployee`'s public services, which include a constructor (lines 12–13) and member functions `earnings` (line 30) and `print` (line 31). Lines 15–28 declare public `get` and `set` functions that manipulate the class's data members (declared in lines 33–37) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`. The `CommissionEmployee` header file specifies that these data members are private, so objects of other classes cannot directly access this data. Declaring data members as private and providing non-private `get` and `set` functions to manipulate and validate the data members helps enforce good software engineering. Member functions `setGrossSales` (defined in lines 56–59 of Fig. 20.5) and `setCommissionRate` (defined in lines 68–71 of Fig. 20.5), for example, validate their arguments before assigning the values to data members `grossSales` and `commissionRate`, respectively.

---

1 // Fig. 20.4: CommissionEmployee.h
2 // CommissionEmployee class definition represents a commission employee.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
```cpp
#include <string> // C++ standard string class
using namespace std;

class CommissionEmployee
{
public:
    CommissionEmployee( const string &, const string &, const string &,
                      double = 0.0, double = 0.0 );
    void setFirstName( const string & ); // set first name
    string getFirstName() const; // return first name
    void setLastName( const string & ); // set last name
    string getLastName() const; // return last name
    void setSocialSecurityNumber( const string & ); // set SSN
    string getSocialSecurityNumber() const; // return SSN
    void setGrossSales( double ); // set gross sales amount
    double getGrossSales() const; // return gross sales amount
    void setCommissionRate( double ); // set commission rate (percentage)
    double getCommissionRate() const; // return commission rate
    double earnings() const; // calculate earnings
    void print() const; // print CommissionEmployee object
private:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales; // gross weekly sales
    double commissionRate; // commission percentage
}; // end class CommissionEmployee
```

Fig. 20.4 | CommissionEmployee class header file. (Part 2 of 2.)

```cpp
// Fig. 20.5: CommissionEmployee.cpp
// Class CommissionEmployee member-function definitions.
#include <iostream>
#include "CommissionEmployee.h" // CommissionEmployee class definition
using namespace std;

// constructor
CommissionEmployee::CommissionEmployee( const string &first, const string &last, const string &ssn,
                                         double sales, double rate )
{
    firstName = first; // should validate
```

Fig. 20.5 | Implementation file for CommissionEmployee class that represents an employee who is paid a percentage of gross sales. (Part 1 of 3.)
Figure 20.5 | Implementation file for `CommissionEmployee` class that represents an employee who is paid a percentage of gross sales. (Part 2 of 3.)
Chapter 20  Object-Oriented Programming: Inheritance

The `CommissionEmployee` constructor definition purposely does not use member-initializer syntax in the first several examples of this section, so that we can demonstrate how private and protected specifiers affect member access in derived classes. As shown in Fig. 20.5, lines 12–14, we assign values to data members `firstName`, `lastName` and `socialSecurityNumber` in the constructor body. Later in this section, we’ll return to using member-initializer lists in the constructors.

We do not validate the values of the constructor’s arguments `first`, `last` and `ssn` before assigning them to the corresponding data members. We certainly could validate the first and last names—perhaps by ensuring that they’re of a reasonable length. Similarly, a social security number could be validated to ensure that it contains nine digits, with or without dashes (e.g., 123-45-6789 or 123456789).

Member function `earnings` (lines 80–83) calculates a `CommissionEmployee`’s earnings. Line 82 multiplies the `commissionRate` by the `grossSales` and returns the result. Member function `print` (lines 86–92) displays the values of a `CommissionEmployee` object’s data members.

Figure 20.6 tests class `CommissionEmployee`. Lines 11–12 instantiate object `employee` of class `CommissionEmployee` and invoke `CommissionEmployee`’s constructor to initialize the object with "Sue" as the first name, "Jones" as the last name, "222-22-2222" as the
// Fig. 20.6: fig20_06.cpp
// Testing class CommissionEmployee.
#include <iostream>
#include <iomanip>
#include "CommissionEmployee.h" // CommissionEmployee class definition
using namespace std;

int main() {
    // instantiate a CommissionEmployee object
    CommissionEmployee employee("Sue", "Jones", "222-22-2222", 10000, .06);

    // set floating-point output formatting
    cout << fixed << setprecision(2);

    // get commission employee data
    cout << "Employee information obtained by get functions: \n"
         << "First name is " << employee.getFirstName() << endl;
    cout << "Last name is " << employee.getLastName() << endl;
    cout << "Social security number is "
         << employee.getSocialSecurityNumber() << endl;
    cout << "Gross sales is " << employee.getGrossSales() << endl;
    cout << "Commission rate is " << employee.getCommissionRate() << endl;

    employee.setGrossSales(8000); // set gross sales
    employee.setCommissionRate(.1); // set commission rate

    cout << "Updated employee information output by print function: \n" << endl;
    employee.print(); // display the new employee information

    // display the employee's earnings
    cout << "Employee's earnings: $" << employee.earnings() << endl;
}

Employee information obtained by get functions:
First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information output by print function:
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 8000.00
commission rate: 0.10

Employee's earnings: $800.00

Fig. 20.6 | CommissionEmployee class test program.
social security number, 10000 as the gross sales amount and .06 as the commission rate.

Lines 19–24 use employee’s get functions to display the values of its data members. Lines 26–27 invoke the object’s member functions setGrossSales and setCommissionRate to change the values of data members grossSales and commissionRate, respectively. Line 31 then calls employee’s print member function to output the updated CommissionEmployee information. Finally, line 34 displays the CommissionEmployee’s earnings, calculated by the object’s earnings member function using the updated values of data members grossSales and commissionRate.

### 20.4.2 Creating a BasePlusCommissionEmployee Class Without Using Inheritance

We now discuss the second part of our introduction to inheritance by creating and testing (a completely new and independent) class BasePlusCommissionEmployee (Figs. 20.7–20.8), which contains a first name, last name, social security number, gross sales amount, commission rate and base salary.

---

```cpp
// Fig. 20.7: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class definition represents an employee
// that receives a base salary in addition to commission.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
using namespace std;

class BasePlusCommissionEmployee
{
public:
    BasePlusCommissionEmployee( const string &, const string &, const string &, double = 0.0, double = 0.0, double = 0.0 );

    void setFirstName( const string & ); // set first name
    string getFirstName() const; // return first name

    void setLastName( const string & ); // set last name
    string getLastName() const; // return last name

    void setSocialSecurityNumber( const string & ); // set SSN
    string getSocialSecurityNumber() const; // return SSN

    void setGrossSales( double ); // set gross sales amount
    double getGrossSales() const; // return gross sales amount

    void setCommissionRate( double ); // set commission rate
    double getCommissionRate() const; // return commission rate

    void setBaseSalary( double ); // set base salary
    double getBaseSalary() const; // return base salary

Fig. 20.7 | BasePlusCommissionEmployee class header file. (Part 1 of 2.)
```
20.4 Relationship between Base Classes and Derived Classes

```cpp
double earnings() const; // calculate earnings
void print() const; // print BasePlusCommissionEmployee object
private:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales; // gross weekly sales
    double commissionRate; // commission percentage
    double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee
```

---

**Fig. 20.7** | BasePlusCommissionEmployee class header file. (Part 2 of 2.)

```cpp
#include <iostream>
#include "BasePlusCommissionEmployee.h"
using namespace std;

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
    const string &first, const string &last, const string &ssn,
    double sales, double rate, double salary )
{
    firstName = first; // should validate
    lastName = last; // should validate
    socialSecurityNumber = ssn; // should validate
    setGrossSales( sales ); // validate and store gross sales
    setCommissionRate( rate ); // validate and store commission rate
    setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

// set first name
void BasePlusCommissionEmployee::setFirstName( const string &first )
{
    firstName = first; // should validate
} // end function setFirstName

// return first name
string BasePlusCommissionEmployee::getFirstName() const
{
    return firstName;
} // end function getFirstName

// set last name
void BasePlusCommissionEmployee::setLastName( const string &last )
{
    lastName = last; // should validate
} // end function setLastName
```

---

**Fig. 20.8** | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 1 of 3.)
37 // return last name
38 string BasePlusCommissionEmployee::getLastName() const
39 {
40     return lastName;
41 } // end function getLastName

44 // set social security number
45 void BasePlusCommissionEmployee::setSocialSecurityNumber(
46     const string &ssn )
47 {
48     socialSecurityNumber = ssn; // should validate
49 } // end function setSocialSecurityNumber

52 // return social security number
53 string BasePlusCommissionEmployee::getSocialSecurityNumber() const
54 {
55     return socialSecurityNumber;
56 } // end function getSocialSecurityNumber

59 // set gross sales amount
60 void BasePlusCommissionEmployee::setGrossSales( double sales )
61 {
62     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
63 } // end function setGrossSales

67 // return gross sales amount
68 double BasePlusCommissionEmployee::getGrossSales() const
69 {
70     return grossSales;
71 } // end function getGrossSales

74 // set commission rate
75 void BasePlusCommissionEmployee::setCommissionRate( double rate )
76 {
77     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
78 } // end function setCommissionRate

82 // set base salary
83 void BasePlusCommissionEmployee::setBaseSalary( double salary )
84 {
85     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
86 } // end function setBaseSalary

Fig. 20.8 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 2 of 3.)
20.4 Relationship between Base Classes and Derived Classes

Defining Class `BasePlusCommissionEmployee`

The `BasePlusCommissionEmployee` header file (Fig. 20.7) specifies class `BasePlusCommissionEmployee`'s public services, which include the `BasePlusCommissionEmployee` constructor (lines 13–14) and member functions `earnings` (line 34) and `print` (line 35). Lines 16–32 declare public `get` and `set` functions for the class's private data members (declared in lines 37–42) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` and `baseSalary`. These variables and member functions encapsulate all the necessary features of a base-salaried commission employee. Note the similarity between this class and class `CommissionEmployee` (Figs. 20.4–20.5)—in this example, we won’t yet exploit that similarity.

Class `BasePlusCommissionEmployee`'s `earnings` member function (defined in lines 94–97 of Fig. 20.8) computes the earnings of a base-salaried commission employee. Line 96 returns the result of adding the employee's base salary to the product of the commission rate and the employee's gross sales.

Testing Class `BasePlusCommissionEmployee`

Figure 20.9 tests class `BasePlusCommissionEmployee`. Lines 11–12 instantiate object `employee` of class `BasePlusCommissionEmployee`, passing "Bob", "Lewis", "333-33-3333", 5000, .04 and 300 to the constructor as the first name, last name, social security number, gross sales, commission rate and base salary, respectively. Lines 19–25 use `BasePlusCommissionEmployee`'s `get` functions to retrieve the values of the object’s data members for output. Line 27 invokes the object’s `setBaseSalary` member function to change the base salary. Member function `setBaseSalary` (Fig. 20.8, lines 82–85) ensures that data member `baseSalary` is not assigned a negative value, because an employee’s base salary cannot
// Fig. 20.9: fig20_09.cpp
// Testing class BasePlusCommissionEmployee.
#include <iostream>
#include <iomanip>
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main()
{
    // instantiate BasePlusCommissionEmployee object
    BasePlusCommissionEmployee employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

    // set floating-point output formatting
    cout << fixed << setprecision( 2 );

    // get commission employee data
    cout << "Employee information obtained by get functions: \n"
    << "\nFirst name is " << employee.getFirstName() << endl;
    cout << "Last name is " << employee.getLastName() << endl;
    cout << "Social security number is "
    << employee.getSocialSecurityNumber() << endl;
    cout << "Gross sales is " << employee.getGrossSales() << endl;
    cout << "Commission rate is " << employee.getCommissionRate() << endl;
    cout << "Base salary is " << employee.getBaseSalary() << endl;
    employee.setBaseSalary( 1000 ); // set base salary

    cout << "Updated employee information output by print function: \n"
    << employee.print() << endl; // display the new employee information

    // display the employee's earnings
    cout << "Employee's earnings: $" << employee.earnings() << endl;
}

Employee information obtained by get functions:
First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information output by print function:
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00

Fig. 20.9 | BasePlusCommissionEmployee class test program.
be negative. Line 31 of Fig. 20.9 invokes the object’s print member function to output the updated BasePlusCommissionEmployee’s information, and line 34 calls member function earnings to display the BasePlusCommissionEmployee’s earnings.

**Exploring the Similarities Between Class BasePlusCommissionEmployee and Class CommissionEmployee**

Most of the code for class BasePlusCommissionEmployee (Figs. 20.7–20.8) is similar, if not identical, to the code for class CommissionEmployee (Figs. 20.4–20.5). For example, in class BasePlusCommissionEmployee, private data members firstName and lastName and member functions setFirstName, getFirstName, setLastName and getLastName are identical to those of class CommissionEmployee. Classes CommissionEmployee and BasePlusCommissionEmployee also both contain private data members socialSecurityNumber, commissionRate and grossSales, as well as get and set functions to manipulate these members. In addition, the BasePlusCommissionEmployee constructor is almost identical to that of class CommissionEmployee, except that BasePlusCommissionEmployee’s constructor also sets the baseSalary. The other additions to class BasePlusCommissionEmployee are private data member baseSalary and member functions setBaseSalary and getBaseSalary. Class BasePlusCommissionEmployee’s print member function is nearly identical to that of class CommissionEmployee, except that BasePlusCommissionEmployee’s print also outputs the value of data member baseSalary.

We literally copied code from class CommissionEmployee and pasted it into class BasePlusCommissionEmployee, then modified class BasePlusCommissionEmployee to include a base salary and member functions that manipulate the base salary. This “copy-and-paste” approach is error prone and time consuming. Worse yet, it can spread many physical copies of the same code throughout a system, creating a code-maintenance nightmare. Is there a way to “absorb” the data members and member functions of a class in a way that makes them part of another class without duplicating code? In the next several examples, we do exactly this, using inheritance.

**Software Engineering Observation 20.3**

Copying and pasting code from one class to another can spread errors across multiple source code files. To avoid duplicating code (and possibly errors), use inheritance, rather than the “copy-and-paste” approach, in situations where you want one class to “absorb” the data members and member functions of another class.

**Software Engineering Observation 20.4**

With inheritance, the common data members and member functions of all the classes in the hierarchy are declared in a base class. When changes are required for these common features, you need to make the changes only in the base class—derived classes then inherit the changes. Without inheritance, changes would need to be made to all the source code files that contain a copy of the code in question.

**20.4.3 Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy**

Now we create and test a new BasePlusCommissionEmployee class (Figs. 20.10–20.11) that derives from class CommissionEmployee (Figs. 20.4–20.5). In this example, a BasePlusCommissionEmployee object is a CommissionEmployee (because inheritance
Chapter 20  Object-Oriented Programming: Inheritance

passes on the capabilities of class `CommissionEmployee`, but class `BasePlusCommissionEmployee` also has data member `baseSalary` (Fig. 20.10, line 23). The colon (:) in line 11 of the class definition indicates inheritance. Keyword `public` indicates the type of inheritance. As a derived class (formed with public inheritance), `BasePlusCommissionEmployee` inherits all the members of class `CommissionEmployee`, except for the constructor—each class provides its own constructors that are specific to the class. (Destructors, too, are not inherited.) Thus, the public services of `BasePlusCommissionEmployee` include its constructor (lines 14–15) and the public member functions inherited from class `CommissionEmployee`—although we cannot see these inherited member functions in `BasePlusCommissionEmployee`'s source code, they're nevertheless a part of derived class `BasePlusCommissionEmployee`. The derived class's public services also include member functions `setBaseSalary`, `getBaseSalary`, `earnings` and `print` (lines 17–21).

```
// Fig. 20.10: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class
// CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class declaration
using namespace std;

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
    BasePlusCommissionEmployee( const string &, const string &, 
        const string &, double = 0.0, double = 0.0, double = 0.0 );

    void setBaseSalary( double ); // set base salary
    double getBaseSalary() const; // return base salary

    double earnings() const; // calculate earnings
    void print() const; // print BasePlusCommissionEmployee object

private:
    double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif
```

**Fig. 20.10** | `BasePlusCommissionEmployee` class definition indicating inheritance relationship with class `CommissionEmployee`.

```
// Fig. 20.11: BasePlusCommissionEmployee.cpp
// Class BasePlusCommissionEmployee member-function definitions.
#include <iostream>
#include "BasePlusCommissionEmployee.h"
using namespace std;

```

**Fig. 20.11** | `BasePlusCommissionEmployee` implementation file: private base-class data cannot be accessed from derived class. (Part 1 of 3.)
20.4 Relationship between Base Classes and Derived Classes

```cpp
// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
    const string &first, const string &last, const string &ssn,
    double sales, double rate, double salary )
    // explicitly call base-class constructor
    : CommissionEmployee( first, last, ssn, sales, rate )
{
    setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
    return baseSalary;
} // end function getBaseSalary

// calculate earnings
double BasePlusCommissionEmployee::earnings() const
{
    // derived class cannot access the base class's private data
    return baseSalary + ( commissionRate * grossSales );
} // end function earnings

// print BasePlusCommissionEmployee object
void BasePlusCommissionEmployee::print() const
{
    // derived class cannot access the base class's private data
    cout << "base-salaried commission employee: " << firstName << ' ' << lastName << "\nsocial security number: " << socialSecurityNumber
    "\ngross sales: " << grossSales
    "\ncommission rate: " << commissionRate
    "\nbase salary: " << baseSalary;
} // end function print
```

Fig. 20.11 | BasePlusCommissionEmployee implementation file: private base-class data cannot be accessed from derived class. (Part 2 of 3.)
Figure 20.11 shows `BasePlusCommissionEmployee`'s member-function implementations. The constructor (lines 8–15) introduces base-class initializer syntax (line 12), which uses a member initializer to pass arguments to the base-class (`CommissionEmployee`) constructor. C++ requires that a derived-class constructor call its base-class constructor to initialize the base-class data members that are inherited into the derived class. Line 12 accomplishes this task by invoking the `CommissionEmployee` constructor by name, passing the constructor's parameters `firstName`, `lastName`, `ssn`, `sales` and `rate` as arguments to initialize base-class data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`. If `BasePlusCommissionEmployee`'s constructor did not invoke class `CommissionEmployee`'s constructor explicitly, C++ would attempt to invoke class `CommissionEmployee`'s default constructor—but the class does not have such a constructor, so the compiler would issue an error. Recall from Chapter 16 that the compiler provides a default constructor with no parameters in any class that does not explicitly include a constructor. However, `CommissionEmployee` does explicitly include a constructor, so a default constructor is not provided, and any attempts to implicitly call `CommissionEmployee`'s default constructor would result in compilation errors.

**Common Programming Error 20.1**

When a derived-class constructor calls a base-class constructor, the arguments passed to the base-class constructor must be consistent with the number and types of parameters specified in one of the base-class constructors; otherwise, a compilation error occurs.

**Performance Tip 20.1**

In a derived-class constructor, initializing member objects and invoking base-class constructors explicitly in the member initializer list prevents duplicate initialization in which a default constructor is called, then data members are modified again in the derived-class constructor’s body.

The compiler generates errors for line 33 of Fig. 20.11 because base class `CommissionEmployee`'s data members `commissionRate` and `grossSales` are private—derived class `BasePlusCommissionEmployee`'s member functions are not allowed to access base class...
CommissionEmployee's private data. We used darker blue text in Fig. 20.11 to indicate erroneous code. The compiler issues additional errors in lines 40–43 of BasePlusCommissionEmployee's print member function for the same reason. As you can see, C++ rigidly enforces restrictions on accessing private data members, so that even a derived class (which is intimately related to its base class) cannot access the base class's private data. [Note: To save space, we show only the error messages from Visual C++ in this example and we removed some of the error messages. The error messages produced by your compiler may differ from those shown here.]

We purposely included the erroneous code in Fig. 20.11 to emphasize that a derived class's member functions cannot access its base class's private data. The errors in BasePlusCommissionEmployee could have been prevented by using the get member functions inherited from class CommissionEmployee. For example, line 33 could have invoked getCommissionRate and getGrossSales to access CommissionEmployee's private data members commissionRate and grossSales, respectively. Similarly, lines 40–43 could have used appropriate get member functions to retrieve the values of the base class's data members. In the next example, we show how using protected data also allows us to avoid the errors encountered in this example.

Including the Base-Class Header File in the Derived-Class Header File with #include

Notice that we #include the base class's header file in the derived class's header file (line 8 of Fig. 20.10). This is necessary for three reasons. First, for the derived class to use the base class's name in line 10, we must tell the compiler that the base class exists—the class definition in CommissionEmployee.h does exactly that.

The second reason is that the compiler uses a class definition to determine the size of an object of that class. A client program that creates an object of a class must #include the class definition in order to enable the compiler to reserve the proper amount of memory for the object. When using inheritance, a derived-class object's size depends on the data members declared explicitly in its class definition and the data members inherited from its direct and indirect base classes. Including the base class's definition in line 8 allows the compiler to determine the memory requirements for the base class's data members that become part of a derived-class object and thus contribute to the total size of the derived-class object.

The last reason for line 8 is to allow the compiler to determine whether the derived class uses the base class's inherited members properly. For example, in the program of Figs. 20.10–20.11, the compiler uses the base-class header file to determine that the data members being accessed by the derived class are private in the base class. Since these are inaccessible to the derived class, the compiler generates errors. The compiler also uses the base class's function prototypes to validate function calls made by the derived class to the inherited base-class functions—you'll see an example of such a function call in Fig. 20.15.

Linking Process in an Inheritance Hierarchy

In Section 16.8, we discussed the linking process for creating an executable GradeBook application. In that example, you saw that the client's object code was linked with the object code for class GradeBook, as well as the object code for any C++ Standard Library classes used in either the client code or in class GradeBook.

The linking process is similar for a program that uses classes in an inheritance hierarchy. The process requires the object code for all classes used in the program and the object code for the direct and indirect base classes of any derived classes used by the pro-
program. Suppose a client wants to create an application that uses class BasePlusCommissionEmployee, which is a derived class of CommissionEmployee (we'll see an example of this in Section 20.4.4). When compiling the client application, the client's object code must be linked with the object code for classes BasePlusCommissionEmployee and CommissionEmployee, because BasePlusCommissionEmployee inherits member functions from its base class CommissionEmployee. The code is also linked with the object code for any C++ Standard Library classes used in class CommissionEmployee, class BasePlusCommissionEmployee or the client code. This provides the program with access to the implementations of all of the functionality that the program may use.

20.4.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Data

To enable class BasePlusCommissionEmployee to directly access CommissionEmployee data members firstName, lastName, socialSecurityNumber, grossSales and commissionRate, we can declare those members as protected in the base class. As we discussed in Section 20.3, a base class's protected members can be accessed by members and friends of the base class and by members and friends of any classes derived from that base class.

Good Programming Practice 20.1
Declare public members first, protected members second and private members last.

Defining Base Class CommissionEmployee with protected Data

Class CommissionEmployee (Figs. 20.12–20.13) now declares data members firstName, lastName, socialSecurityNumber, grossSales and commissionRate as protected (Fig. 20.12, lines 32–37) rather than private. The member-function implementations in Fig. 20.13 are identical to those in Fig. 20.5.

```
// Fig. 20.12: CommissionEmployee.h
// CommissionEmployee class definition with protected data.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
using namespace std;

class CommissionEmployee
{
public:
    CommissionEmployee( const string &, const string &, const string &,
                        double = 0.0, double = 0.0 );
    void setFirstName( const string & ); // set first name
    string getFirstName() const; // return first name

Fig. 20.12 | CommissionEmployee class definition that declares protected data to allow access by derived classes. (Part 1 of 2.)
```
void setLastName( const string & ); // set last name
string getLastName() const; // return last name

void setSocialSecurityNumber( const string & ); // set SSN
string getSocialSecurityNumber() const; // return SSN

void setGrossSales( double ); // set gross sales amount
double getGrossSales() const; // return gross sales amount

void setCommissionRate( double ); // set commission rate
double getCommissionRate() const; // return commission rate

double earnings() const; // calculate earnings
void print() const; // print CommissionEmployee object

protected:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales; // gross weekly sales
double commissionRate; // commission percentage

}; // end class CommissionEmployee

#endif

// Fig. 20.13: CommissionEmployee.cpp
// Class CommissionEmployee member-function definitions.
#include <iostream>
#include "CommissionEmployee.h" // CommissionEmployee class definition
using namespace std;

// constructor
CommissionEmployee::CommissionEmployee( 
    const string &first, const string &last, const string &ssn,
    double sales, double rate )
{
    firstName = first; // should validate
    lastName = last; // should validate
    socialSecurityNumber = ssn; // should validate
    setGrossSales( sales ); // validate and store gross sales
    setCommissionRate( rate ); // validate and store commission rate
} // end CommissionEmployee constructor

// set first name
void CommissionEmployee::setFirstName( const string &first )
{
    firstName = first; // should validate
} // end function setFirstName

Fig. 20.12 | CommissionEmployee class definition that declares protected data to allow access by derived classes. (Part 2 of 2.)

Fig. 20.13 | CommissionEmployee class with protected data. (Part 1 of 3.)
// return first name
string CommissionEmployee::getFirstName() const
{
    return firstName;
} // end function getFirstName

// set last name
void CommissionEmployee::setLastName( const string &last )
{
    lastName = last; // should validate
} // end function setLastName

// return last name
string CommissionEmployee::getLastName() const
{
    return lastName;
} // end function getLastName

// set social security number
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
    socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string CommissionEmployee::getSocialSecurityNumber() const
{
    return socialSecurityNumber;
} // end function getSocialSecurityNumber

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
} // end function setGrossSales

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
    return grossSales;
} // end function getGrossSales

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
20.4 Relationship between Base Classes and Derived Classes

Figure 20.14

```cpp
// Fig. 20.14: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class
// CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class declaration
using namespace std;

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
    BasePlusCommissionEmployee( const string &, const string &,
                                  const string &, double = 0.0, double = 0.0, double = 0.0 );
};
```

Figure 20.13 | CommissionEmployee class with protected data. (Part 3 of 3.)

Modify Derived Class `BasePlusCommissionEmployee`

The version of class `BasePlusCommissionEmployee` in Figs. 20.14–20.15 inherits from class `CommissionEmployee` in Figs. 20.12–20.13. Objects of class `BasePlusCommissionEmployee` can access inherited data members that are declared protected in class `CommissionEmployee` (i.e., data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`). As a result, the compiler does not generate errors when compiling the `BasePlusCommissionEmployee` earnings and print member-function definitions in Fig. 20.15 (lines 30–34 and 37–45, respectively). This shows the special privileges that a derived class is granted to access protected base-class data members. Objects of a derived class also can access protected members in any of that derived class's indirect base classes.

```cpp
return commissionRate;
} // end function getCommissionRate

// calculate earnings
double CommissionEmployee::earnings() const
{
    return commissionRate * grossSales;
} // end function earnings

// print CommissionEmployee object
void CommissionEmployee::print() const
{
    cout << "commission employee: " << firstName << ' ' << lastName
         << "\nsocial security number: " << socialSecurityNumber
         << "\ngross sales: " << grossSales
         << "\ncommission rate: " << commissionRate;
} // end function print
```

Figure 20.13 | CommissionEmployee class with protected data. (Part 3 of 3.)

Modify Derived Class `BasePlusCommissionEmployee`

The version of class `BasePlusCommissionEmployee` in Figs. 20.14–20.15 inherits from class `CommissionEmployee` in Figs. 20.12–20.13. Objects of class `BasePlusCommissionEmployee` can access inherited data members that are declared protected in class `CommissionEmployee` (i.e., data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`). As a result, the compiler does not generate errors when compiling the `BasePlusCommissionEmployee` earnings and print member-function definitions in Fig. 20.15 (lines 30–34 and 37–45, respectively). This shows the special privileges that a derived class is granted to access protected base-class data members. Objects of a derived class also can access protected members in any of that derived class’s indirect base classes.

```cpp
# ifndef BASEPLUS_H
# define BASEPLUS_H

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class declaration
using namespace std;

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
    BasePlusCommissionEmployee( const string &, const string &,
                                  const string &, double = 0.0, double = 0.0, double = 0.0 );
};
```

Figure 20.14 | `BasePlusCommissionEmployee` class header file. (Part 1 of 2.)
void setBaseSalary( double ); // set base salary
double getBaseSalary() const; // return base salary
double earnings() const; // calculate earnings
void print() const; // print BasePlusCommissionEmployee object
private:
double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

Fig. 20.14 | BasePlusCommissionEmployee class header file. (Part 2 of 2.)

// Fig. 20.15: BasePlusCommissionEmployee.cpp
// Class BasePlusCommissionEmployee member-function definitions.
#include <iostream>
#include "BasePlusCommissionEmployee.h"
using namespace std;

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(  
    const string &first, const string &last, const string &ssn,  
    double sales, double rate, double salary )  
    : CommissionEmployee( first, last, ssn, sales, rate )  
{} // end BasePlusCommissionEmployee constructor

// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )  
{}  
// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const  
{}  
// calculate earnings
double BasePlusCommissionEmployee::earnings() const  
{}  

Fig. 20.15 | BasePlusCommissionEmployee implementation file for
BasePlusCommissionEmployee class that inherits protected data from CommissionEmployee.  
(Part 1 of 2.)
20.4 Relationship between Base Classes and Derived Classes

Class `BasePlusCommissionEmployee` does not inherit class `CommissionEmployee`'s constructor. However, class `BasePlusCommissionEmployee`'s constructor (Fig. 20.15, lines 8–15) calls class `CommissionEmployee`'s constructor explicitly with member initializer syntax (line 12). Recall that `BasePlusCommissionEmployee`'s constructor must explicitly call the constructor of class `CommissionEmployee`, because `CommissionEmployee` does not contain a default constructor that could be invoked implicitly.

**Testing the Modified `BasePlusCommissionEmployee` Class**

Figure 20.16 uses a `BasePlusCommissionEmployee` object to perform the same tasks that Fig. 20.9 performed on an object of the first version of class `BasePlusCommissionEmployee` (Figs. 20.7–20.8). The code and outputs of the two programs are identical. We created the first class `BasePlusCommissionEmployee` without using inheritance and created this version of `BasePlusCommissionEmployee` using inheritance; however, both classes provide the same functionality. The code for class `BasePlusCommissionEmployee` (i.e., the header and implementation files), which is 71 lines, is considerably shorter than the code for the noninherited version of the class, which is 152 lines, because the inherited version absorbs part of its functionality from `CommissionEmployee`, whereas the noninherited version does not absorb any functionality. Also, there is now only one copy of the `CommissionEmployee` functionality declared and defined in class `CommissionEmployee`. This makes the source code easier to maintain, modify and debug, because the source code related to a `CommissionEmployee` exists only in the files of Figs. 20.12–20.13.

---

```cpp
// Fig. 20.16: fig20_16.cpp
// Testing class BasePlusCommissionEmployee.
#include <iostream>
#include <iomanip>
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main()
{
    // print BasePlusCommissionEmployee object
    void BasePlusCommissionEmployee::print() const
    {
        // can access protected data of base class
        cout << "base-salaried commission employee: " << firstName << ' ' << lastName << "social security number: " << socialSecurityNumber << "gross sales: " << grossSales << "commission rate: " << commissionRate << "base salary: " << baseSalary;
    }
}

Fig. 20.15 | BasePlusCommissionEmployee implementation file for BasePlusCommissionEmployee class that inherits protected data from CommissionEmployee. (Part 2 of 2.)
Fig. 20.16 | protected base-class data can be accessed from derived class. (Part 1 of 2.)
```
In this example, we declared base-class data members as protected, so derived classes can modify the data directly. Inheriting protected data members slightly increases performance, because we can directly access the members without incurring the overhead of calls to set or get member functions. In most cases, however, it's better to use private data members to encourage proper software engineering, and leave code optimization issues to the compiler. Your code will be easier to maintain, modify and debug.
Using protected data members creates two serious problems. First, the derived-class object does not have to use a member function to set the value of the base class’s protected data member. An invalid value can easily be assigned to the protected data member, thus leaving the object in an inconsistent state—e.g., with CommissionEmployee’s data member grossSales declared as protected, a derived-class object can assign a negative value to grossSales. The second problem with using protected data members is that derived-class member functions are more likely to be written so that they depend on the base-class implementation. Derived classes should depend only on the base-class services (i.e., non-private member functions) and not on the base-class implementation. With protected data members in the base class, if the base-class implementation changes, we may need to modify all derived classes of that base class. For example, if for some reason we were to change the names of data members firstName and lastName to first and last, then we’d have to do so for all occurrences in which a derived class references these base-class data members directly. Such software is said to be fragile or brittle, because a small change in the base class can “break” derived-class implementation. You should be able to change the base-class implementation while still providing the same services to derived classes. (Of course, if the base-class services change, we must reimplement our derived classes—good object-oriented design attempts to prevent this.)

**Software Engineering Observation 20.5**

It’s appropriate to use the protected access specifier when a base class should provide a service (i.e., a member function) only to its derived classes and friends.

**Software Engineering Observation 20.6**

Declaring base-class data members private (as opposed to declaring them protected) enables you to change the base-class implementation without having to change derived-class implementations.

**Error-Prevention Tip 20.1**

When possible, avoid including protected data members in a base class. Rather, include non-private member functions that access private data members, ensuring that the object maintains a consistent state.

### 20.4.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Data

We now reexamine our hierarchy once more, this time using the best software engineering practices. Class CommissionEmployee (Figs. 20.17–20.18) now declares data members firstName, lastName, socialSecurityNumber, grossSales and commissionRate as private (Fig. 20.17, lines 32–37) and provides public member functions setFirstName, getFirstName, setLastName, getLastName, setSocialSecurityNumber, getSocialSecurityNumber, setGrossSales, getGrossSales, setCommissionRate, getCommissionRate, earnings and print for manipulating these values. If we decide to change the data member names, the earnings and print definitions will not require modification—only the definitions of the get and set member functions that directly manipulate the data members will need to change. These changes occur solely within the base class—no changes to the derived class are needed. Localizing the effects of changes like this is a good
software engineering practice. Derived class BasePlusCommissionEmployee (Figs. 20.19–20.20) inherits CommissionEmployee's member functions and can access the private base-class members via the inherited non-private member functions.

```cpp
// Fig. 20.17: CommissionEmployee.h
// CommissionEmployee class definition with good software engineering.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
using namespace std;

class CommissionEmployee
{
public:
    CommissionEmployee( const string &, const string &, const string &,
                         double = 0.0, double = 0.0 );

    void setFirstName( const string & ); // set first name
    string getFirstName() const; // return first name

    void setLastName( const string & ); // set last name
    string getLastName() const; // return last name

    void setSocialSecurityNumber( const string & ); // set SSN
    string getSocialSecurityNumber() const; // return SSN

    void setGrossSales( double ); // set gross sales amount
    double getGrossSales() const; // return gross sales amount

    void setCommissionRate( double ); // set commission rate
    double getCommissionRate() const; // return commission rate

    double earnings() const; // calculate earnings
    void print() const; // print CommissionEmployee object

private:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales; // gross weekly sales
    double commissionRate; // commission percentage
}; // end class CommissionEmployee
#endif
```

**Fig. 20.17** | CommissionEmployee class defined using good software engineering practices.

In the CommissionEmployee constructor implementation (Fig. 20.18, lines 8–15), we use member initializers (line 11) to set the values of members firstName, lastName and socialSecurityNumber. We show how derived-class BasePlusCommissionEmployee (Figs. 20.19–20.20) can invoke non-private base-class member functions (setFirstName, getFirstName, setLastName, getLastName, setSocialSecurityNumber and getSocialSecurityNumber) to manipulate these data members.
Performance Tip 20.2

Using a member function to access a data member’s value can be slightly slower than accessing the data directly. However, today’s optimizing compilers are carefully designed to perform many optimizations implicitly (such as inlining set and get member-function calls). You should write code that adheres to proper software engineering principles, and leave optimization to the compiler. A good rule is, “Do not second-guess the compiler.”

```cpp
// Fig. 20.18: CommissionEmployee.cpp
// Class CommissionEmployee member-function definitions.
#include <iostream>
#include "CommissionEmployee.h" // CommissionEmployee class definition
using namespace std;

// constructor
CommissionEmployee::CommissionEmployee(
    const string &first, const string &last, const string &ssn,
    double sales, double rate ) : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
{
    setGrossSales( sales ); // validate and store gross sales
    setCommissionRate( rate ); // validate and store commission rate
} // end CommissionEmployee constructor

// set first name
void CommissionEmployee::setFirstName( const string &first )
{
    firstName = first; // should validate
} // end function setFirstName

// return first name
string CommissionEmployee::getFirstName() const
{
    return firstName;
} // end function getFirstName

// set last name
void CommissionEmployee::setLastName( const string &last )
{
    lastName = last; // should validate
} // end function setLastName

// return last name
string CommissionEmployee::getLastname() const
{
    return lastName;
} // end function getLastname

// set social security number
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
```

Fig. 20.18 | CommissionEmployee class implementation file: CommissionEmployee class uses member functions to manipulate its private data. (Part 1 of 2.)
Chapter 20  Object-Oriented Programming: Inheritance

Fig. 20.18  |  CommissionEmployee class implementation file: CommissionEmployee class uses member functions to manipulate its private data. (Part 2 of 2.)

Class BasePlusCommissionEmployee (Figs. 20.19–20.20) has several changes to its member-function implementations (Fig. 20.20) that distinguish it from the previous version
of the class (Figs. 20.14–20.15). Member functions earnings (Fig. 20.20, lines 30–33) and print (lines 36–44) each invoke member function getBaseSalary to obtain the base salary value, rather than accessing baseSalary directly. This insulates earnings and print from potential changes to the implementation of data member baseSalary. For example, if we decide to rename data member baseSalary or change its type, only member functions setBaseSalary and getBaseSalary will need to change.

```cpp
// Fig. 20.19: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class declaration
using namespace std;

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
    BasePlusCommissionEmployee( const string &, const string &, const string &, double = 0.0, double = 0.0, double = 0.0 );

    void setBaseSalary( double ); // set base salary
    double getBaseSalary() const; // return base salary

    double earnings() const; // calculate earnings
    void print() const; // print BasePlusCommissionEmployee object

private:
    double const baseSalary; // base salary
}; // end class BasePlusCommissionEmployee
#endif
```

**Fig. 20.19** | BasePlusCommissionEmployee class header file.

```cpp
// Fig. 20.20: BasePlusCommissionEmployee.cpp
// Class BasePlusCommissionEmployee member-function definitions.
#include <iostream>
#include "BasePlusCommissionEmployee.h"
using namespace std;

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
    const string &first, const string &last, const string &ssn,
    double sales, double rate, double salary )
    : explicitly call base-class constructor
{
    CommissionEmployee( first, last, ssn, sales, rate )
```

**Fig. 20.20** | BasePlusCommissionEmployee class that inherits from class CommissionEmployee but cannot directly access the class’s private data. (Part 1 of 2.)
Class BasePlusCommissionEmployee’s earnings function (Fig. 20.20, lines 30–33) redefines class CommissionEmployee’s earnings member function (Fig. 20.18, lines 78–81) to calculate the earnings of a base-salaried commission employee. Class BasePlusCommissionEmployee’s version of earnings obtains the portion of the employee’s earnings based on commission alone by calling base-class CommissionEmployee’s earnings function with the expression CommissionEmployee::earnings() (Fig. 20.20, line 32). BasePlusCommissionEmployee’s earnings function then adds the base salary to this value to calculate the total earnings of the employee. Note the syntax used to invoke a redefined base-class member function from a derived class—place the base-class name and the binary scope resolution operator (::) before the base-class member-function name. This member-function invocation is a good software engineering practice: If an object’s member function performs the actions needed by another object, call that member function rather than duplicating its code body. By having BasePlusCommissionEmployee’s earnings function invoke CommissionEmployee’s earnings function to calculate part of a BasePlusCommissionEmployee object’s earnings, we avoid duplicating the code and reduce code-maintenance problems.
Similarly, BasePlusCommissionEmployee’s print function (Fig. 20.20, lines 36–44) redefines class CommissionEmployee’s print function (Fig. 20.18, lines 84–91) to output the appropriate base-salaried commission employee information. The new version displays part of a BasePlusCommissionEmployee object’s information (i.e., the string "commission employee" and the values of class CommissionEmployee’s private data members) by calling CommissionEmployee’s print member function with the qualified name CommissionEmployee::print() (Fig. 20.20, line 41). BasePlusCommissionEmployee’s print function then outputs the remainder of a BasePlusCommissionEmployee object’s information (i.e., the value of class BasePlusCommissionEmployee’s base salary).

Figure 20.21 performs the same manipulations on a BasePlusCommissionEmployee object as did Fig. 20.9 and Fig. 20.16 on objects of classes CommissionEmployee and BasePlusCommissionEmployee, respectively. Although each “base-salaried commission employee” class behaves identically, class BasePlusCommissionEmployee is the best engineered. By using inheritance and by calling member functions that hide the data and ensure consistency, we’ve efficiently and effectively constructed a well-engineered class.

```cpp
1 // Fig. 20.21: fig20_21.cpp
2 // Testing class BasePlusCommissionEmployee.
3 #include <iostream>
4 #include <iomanip>
5 #include "BasePlusCommissionEmployee.h"
6 using namespace std;
7
8 int main()
9 {
10   // instantiate BasePlusCommissionEmployee object
11   BasePlusCommissionEmployee employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13   // set floating-point output formatting
14   cout << fixed << setprecision( 2 );
15
16   // get commission employee data
17   cout << "Employee information obtained by get functions: \n"
18       << "\nFirst name is " << employee.getFirstName()
19       << "\nLast name is " << employee.getLastName()
20       << "\nSocial security number is "
21       << employee.getSocialSecurityNumber()
22       << "\nGross sales is " << employee.getGrossSales()
23       << "\nCommission rate is " << employee.getCommissionRate()
24       << "\nBase salary is " << employee.getBaseSalary() << endl;
```

**Common Programming Error 20.2**

When a base-class member function is redefined in a derived class, the derived-class version often calls the base-class version to do additional work. Failure to use the :: operator prefixed with the name of the base class when referencing the base class’s member function causes infinite recursion, because the derived-class member function would then call itself.
In this section, you saw an evolutionary set of examples that was carefully designed to teach key capabilities for good software engineering with inheritance. You learned how to create a derived class using inheritance, how to use protected base-class members to enable a derived class to access inherited base-class data members and how to redefine base-class functions to provide versions that are more appropriate for derived-class objects. In addition, you learned how to apply software engineering techniques from Chapters 17–18 and this chapter to create classes that are easy to maintain, modify and debug.

20.5 Constructors and Destructors in Derived Classes

As we explained in the preceding section, instantiating a derived-class object begins a chain of constructor calls in which the derived-class constructor, before performing its own tasks, invokes its direct base class’s constructor either explicitly (via a base-class member initializer) or implicitly (calling the base class’s default constructor). Similarly, if the base class is derived from another class, the base-class constructor is required to invoke the constructor of the next class up in the hierarchy, and so on. The last constructor called in this chain is the constructor of the class at the base of the hierarchy, whose body actually finishes executing first. The original derived-class constructor’s body finishes executing last.

Each base-class constructor initializes the base-class data members that the derived-class...
object inherits. For example, consider the CommissionEmployee/BasePlusCommissionEmployee hierarchy from Figs. 20.17–20.20. When a program creates an object of class BasePlusCommissionEmployee, the CommissionEmployee constructor is called. Since class CommissionEmployee is at the base of the hierarchy, its constructor executes, initializing the private data members of CommissionEmployee that are part of the BasePlusCommissionEmployee object. When CommissionEmployee’s constructor completes execution, it returns control to BasePlusCommissionEmployee’s constructor, which initializes the BasePlusCommissionEmployee object’s baseSalary.

Software Engineering Observation 20.7
When a program creates a derived-class object, the derived-class constructor immediately calls the base-class constructor, the base-class constructor’s body executes, then the derived class’s member initializers execute and finally the derived-class constructor’s body executes. This process cascades up the hierarchy if it contains more than two levels.

When a derived-class object is destroyed, the program calls that object’s destructor. This begins a chain (or cascade) of destructor calls in which the derived-class destructor and the destructors of the direct and indirect base classes and the classes’ members execute in reverse of the order in which the constructors executed. When a derived-class object’s destructor is called, the destructor performs its task, then invokes the destructor of the next base class up the hierarchy. This process repeats until the destructor of the final base class at the top of the hierarchy is called. Then the object is removed from memory.

Software Engineering Observation 20.8
Suppose that we create an object of a derived class where both the base class and the derived class contain (via composition) objects of other classes. When an object of that derived class is created, first the constructors for the base class’s member objects execute, then the base-class constructor executes, then the constructors for the derived class’s member objects execute, then the derived class’s constructor executes. Destructors for derived-class objects are called in the reverse of the order in which their corresponding constructors are called.

Base-class constructors, destructors and overloaded assignment operators (see Chapter 19) are not inherited by derived classes. Derived-class constructors, destructors and overloaded assignment operators, however, can call base-class constructors, destructors and overloaded assignment operators.

Our next example defines class CommissionEmployee (Figs. 20.22–20.23) and class BasePlusCommissionEmployee (Figs. 20.24–20.25) with constructors and destructors that each print a message when invoked. As you’ll see in the output in Fig. 20.26, these messages demonstrate the order in which the constructors and destructors are called for objects in an inheritance hierarchy.

```c++
// Fig. 20.22: CommissionEmployee.h
// CommissionEmployee class definition represents a commission employee.
#ifndef COMMISSION_H
#define COMMISSION_H
#define COMMISSION_H
```
#include <string> // C++ standard string class
using namespace std;

class CommissionEmployee
{
public:
    CommissionEmployee( const string &, const string &, const string &, double sales = 0.0, double rate = 0.0 );
    ~CommissionEmployee(); // destructor

    void setFirstName( const string & ); // set first name
    string getFirstName() const; // return first name

    void setLastName( const string & ); // set last name
    string getLastName() const; // return last name

    void setSocialSecurityNumber( const string & ); // set SSN
    string getSocialSecurityNumber() const; // return SSN

    void setGrossSales( double ); // set gross sales amount
    double getGrossSales() const; // return gross sales amount

    void setCommissionRate( double ); // set commission rate
    double getCommissionRate() const; // return commission rate

    double earnings() const; // calculate earnings
    void print() const; // print CommissionEmployee object

private:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales; // gross weekly sales
    double commissionRate; // commission percentage
}; // end class CommissionEmployee

Fig. 20.22 | CommissionEmployee class header file. (Part 2 of 2.)

Fig. 20.23 | CommissionEmployee's constructor and destructor output text. (Part 1 of 3.)
setGrossSales( sales ); // validate and store gross sales
setCommissionRate( rate ); // validate and store commission rate

cout << "CommissionEmployee constructor: " << endl;
print();
cout << "\n\n";
} // end CommissionEmployee constructor

// destructor
CommissionEmployee::~CommissionEmployee()
{
    cout << "CommissionEmployee destructor: " << endl;
    print();
cout << "\n\n";
} // end CommissionEmployee destructor

// set first name
void CommissionEmployee::setFirstName( const string &first )
{
    firstName = first; // should validate
} // end function setFirstName

// return first name
string CommissionEmployee::getFirstName() const
{
    return firstName;
} // end function getFirstName

// set last name
void CommissionEmployee::setLastName( const string &last )
{
    lastName = last; // should validate
} // end function setLastName

// return last name
string CommissionEmployee::getLastName() const
{
    return lastName;
} // end function getLastName

// set social security number
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
    socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string CommissionEmployee::getSocialSecurityNumber() const
{
    return socialSecurityNumber;
} // end function getSocialSecurityNumber
// set gross sales amount
void CommissionEmployee::setGrossSales(double sales)
{
    grossSales = (sales < 0.0) ? 0.0 : sales;
} // end function setGrossSales

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
    return grossSales;
} // end function getGrossSales

// set commission rate
void CommissionEmployee::setCommissionRate(double rate)
{
    commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
    return commissionRate;
} // end function getCommissionRate

// calculate earnings
double CommissionEmployee::earnings() const
{
    return getCommissionRate() * getGrossSales();
} // end function earnings

// print CommissionEmployee object
void CommissionEmployee::print() const
{
    cout << "commission employee: "
        << getFirstName() << '"' << getLastName()
        << "\nsocial security number: " << getSocialSecurityNumber()
        << "\ngross sales: " << getGrossSales()
        << "\ncommission rate: " << getCommissionRate();
} // end function print

---

// Fig. 20.23 | CommissionEmployee's constructor and destructor output text. (Part 3 of 3.)

---

// Fig. 20.24: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class declaration

using namespace std;

---

// Fig. 20.24 | BasePlusCommissionEmployee class header file. (Part 1 of 2.)
class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
    BasePlusCommissionEmployee(const string &, const string &, const string &, double = 0.0, double = 0.0, double = 0.0);
    ~BasePlusCommissionEmployee(); // destructor
    void setBaseSalary(double); // set base salary
    double getBaseSalary() const; // return base salary
    double earnings() const; // calculate earnings
    void print() const; // print BasePlusCommissionEmployee object
private:
    double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

// Fig. 20.24 | BasePlusCommissionEmployee class header file. (Part 2 of 2.)

#include <iostream>
#include "BasePlusCommissionEmployee.h"
using namespace std;

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee( const string &first, const string &last, const string &ssn, double sales, double rate, double salary ) :
    CommissionEmployee( first, last, ssn, sales, rate )
{
    setBaseSalary( salary ); // validate and store base salary
    cout << "BasePlusCommissionEmployee constructor: " << endl;
    print();
    cout << "\n\n";
} // end BasePlusCommissionEmployee constructor

// destructor
BasePlusCommissionEmployee::~BasePlusCommissionEmployee()
{
    cout << "BasePlusCommissionEmployee destructor: " << endl;
    print();
    cout << "\n\n";
} // end BasePlusCommissionEmployee destructor

// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
In this example, we modified the CommissionEmployee constructor (lines 8–19 of Fig. 20.23) and included a CommissionEmployee destructor (lines 22–27), each of which outputs a line of text upon its invocation. We also modified the BasePlusCommissionEmployee constructor (lines 8–19 of Fig. 20.25) and included a BasePlusCommissionEmployee destructor (lines 22–27), each of which outputs a line of text upon its invocation.

Figure 20.26 demonstrates the order in which constructors and destructors are called for objects of classes that are part of an inheritance hierarchy. Function main instantiates CommissionEmployee object employee1 (lines 15–16) in a separate block inside main (lines 14–17). The object goes in and out of scope—the end of the block is reached immediately after the object is created—so both the CommissionEmployee constructor and destructor are called. Next, lines 20–21 instantiate BasePlusCommissionEmployee object employee2. This invokes the CommissionEmployee constructor to display outputs with values passed from the BasePlusCommissionEmployee constructor, then the output specified in the BasePlusCommissionEmployee constructor is performed. Lines 24–25 then instantiate BasePlusCommissionEmployee object employee3. Again, the CommissionEmployee and BasePlusCommissionEmployee constructors are both called. In each case, the body of the CommissionEmployee constructor executes before the body of the BasePlusCommissionEmployee constructor executes. When the end of main is reached, the destructors are called for objects employee2 and employee3. But, because destructors are called in the reverse order of their corresponding constructors, the BasePlusCommissionEmployee destructor and CommissionEmployee destructor are called (in that order) for object employee3, then
the BasePlusCommissionEmployee and CommissionEmployee destructors are called (in that order) for object employee2.

```cpp
// Fig. 20.26: fig20_26.cpp
// Display order in which base-class and derived-class constructors
// and destructors are called.
#include <iostream>
#include <iomanip>
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main()
{
    // set floating-point output formatting
    cout << fixed << setprecision( 2 );

    { // begin new scope
        CommissionEmployee employee1("Bob", "Lewis", "333-33-3333", 5000, .04);
    } // end scope

    cout << endl;
    BasePlusCommissionEmployee employee2("Lisa", "Jones", "555-55-5555", 2000, .06, 800);
    cout << endl;
    BasePlusCommissionEmployee employee3("Mark", "Sands", "888-88-8888", 8000, .15, 2000);
    cout << endl;
} // end main
```

---

**CommissionEmployee constructor:**
commission employee: Bob Lewis  
social security number: 333-33-3333  
gross sales: 5000.00  
commission rate: 0.04

**CommissionEmployee destructor:**  
commission employee: Bob Lewis  
social security number: 333-33-3333  
gross sales: 5000.00  
commission rate: 0.04

**CommissionEmployee constructor:**
commission employee: Lisa Jones  
social security number: 555-55-5555  
gross sales: 2000.00  
commission rate: 0.06

**BasePlusCommissionEmployee constructor:**  
base-salaried commission employee: Lisa Jones  
social security number: 555-55-5555

---

**Fig. 20.26**  
Constructor and destructor call order. (Part 1 of 2.)
Chapter 20  Object-Oriented Programming: Inheritance

20.6 **public, protected and private Inheritance**

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. Use of protected and private inheritance is rare, and each should be used only with great care; we normally use public inheritance in this book. Figure 20.27 summarizes for each type of inheritance the accessibility of base-class members in a derived class. The first column contains the base-class access specifiers.
When deriving a class from a public base class, public members of the base class become public members of the derived class, and protected members of the base class become protected members of the derived class. A base class’s private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

When deriving from a protected base class, public and protected members of the base class become protected members of the derived class. When deriving from a private base class, public and protected members of the base class become private members (e.g., the functions become utility functions) of the derived class. Private and protected inheritance are not is-a relationships.

20.7 Software Engineering with Inheritance

In this section, we discuss the use of inheritance to customize existing software. When we use inheritance to create a new class from an existing one, the new class inherits the data members and member functions of the existing class, as described in Fig. 20.27. We can customize the new class to meet our needs by including additional members and by redefining base-class members. The derived-class programmer does this in C++ without accessing the base class’s source code. The derived class must be able to link to the base class’s object code. This powerful capability is attractive to independent software vendors (ISVs).
ISVs can develop proprietary classes for sale or license and make these classes available to users in object-code format. Users then can derive new classes from these library classes rapidly and without accessing the ISVs’ proprietary source code. All the ISVs need to supply with the object code are the header files.

Sometimes it’s difficult for students to appreciate the scope of problems faced by designers who work on large-scale software projects in industry. People experienced with such projects say that effective software reuse improves the software development process. Object-oriented programming facilitates software reuse, thus shortening development times and enhancing software quality.

The availability of substantial and useful class libraries delivers the maximum benefits of software reuse through inheritance. Just as shrink-wrapped software produced by independent software vendors became an explosive-growth industry with the arrival of the personal computer, interest in the creation and sale of class libraries is growing exponentially. Application designers build their applications with these libraries, and library designers are rewarded by having their libraries included with the applications. The standard C++ libraries that are shipped with C++ compilers tend to be rather general purpose and limited in scope. However, there is massive worldwide commitment to the development of class libraries for a huge variety of applications arenas.

**Software Engineering Observation 20.9**

At the design stage in an object-oriented system, the designer often determines that certain classes are closely related. The designer should “factor out” common attributes and behaviors and place these in a base class, then use inheritance to form derived classes, endowing them with capabilities beyond those inherited from the base class.

**Software Engineering Observation 20.10**

The creation of a derived class does not affect its base class’s source code. Inheritance preserves the integrity of a base class.

**Performance Tip 20.3**

If classes produced through inheritance are larger than they need to be (i.e., contain too much functionality), memory and processing resources might be wasted. Inherit from the class whose functionality is “closest” to what’s needed.

Reading derived-class definitions can be confusing, because inherited members are not shown physically in the derived classes, but nevertheless are present. A similar problem exists when documenting derived-class members.

**20.8 Wrap-Up**

This chapter introduced inheritance—the ability to create a class by absorbing an existing class’s data members and member functions and embellishing them with new capabilities. Through a series of examples using an employee inheritance hierarchy, you learned the notions of base classes and derived classes and used public inheritance to create a derived class that inherits members from a base class. The chapter introduced the access specifier protected—derived-class member functions can access protected base-class members. You learned how to access redefined base-class members by qualifying their names with the base-class name and binary scope resolution operator (::). You also saw the order in
which constructors and destructors are called for objects of classes that are part of an inheritance hierarchy. Finally, we explained the three types of inheritance—public, protected, and private—and the accessibility of base-class members in a derived class when using each type.

In Chapter 21, we build on our discussion of inheritance by introducing polymorphism—an object-oriented concept that enables us to write programs that handle, in a more general manner, objects of a wide variety of classes related by inheritance. After studying Chapter 21, you’ll be familiar with classes, objects, encapsulation, inheritance and polymorphism—the essential concepts of object-oriented programming.

Summary

Section 20.1 Introduction
• Software reuse reduces program development time and cost.

Section 20.2 Base Classes and Derived Classes
• Inheritance is a form of software reuse in which you create a class that absorbs an existing class’s data and behaviors and enhances them with new capabilities. The existing class is called the base class, and the new class is referred to as the derived class.
• A direct base class is the one from which a derived class explicitly inherits. An indirect base class is inherited from two or more levels up the class hierarchy.
• With single inheritance, a class is derived from one base class. With multiple inheritance, a class inherits from multiple (possibly unrelated) base classes.
• A derived class represents a more specialized group of objects. Typically, a derived class contains behaviors inherited from its base class plus additional behaviors. A derived class can also customize behaviors inherited from the base class.
• Every object of a derived class is also an object of that class’s base class. However, a base-class object is not an object of that class’s derived classes.
• The is-a relationship represents inheritance. In an is-a relationship, an object of a derived class also can be treated as an object of its base class.
• The has-a relationship represents composition—an object contains one or more objects of other classes as members, but does not disclose their behavior directly in its interface.
• A derived class cannot access the private members of its base class directly. A derived class can access the public and protected members of its base class directly.
• A derived class can effect state changes in private base-class members, but only through non-private member functions provided in the base class and inherited into the derived class.
• A base-class member function can be redefined in a derived class.
• Single-inheritance relationships form treelike hierarchical structures.
• It’s possible to treat base-class objects and derived-class objects similarly; the commonality shared between the object types is expressed in the base class’s data members and member functions.

Section 20.3 protected Members
• A base class’s public members are accessible anywhere that the program has a handle to an object of that base class or to an object of one of that base class’s derived classes—or, when using the binary scope resolution operator, whenever the class’s name is in scope.
• A base class’s private members are accessible only within the base class or from its friends.
• A base class’s protected members can be accessed by members and friends of that base class and by members and friends of any classes derived from that base class.
• When a derived-class member function redefines a base-class member function, the base-class member function can be accessed from the derived class by qualifying the base-class member function name with the base-class name and the binary scope resolution operator (::).

Section 20.5 Constructors and Destructors in Derived Classes
• When an object of a derived class is instantiated, the base class’s constructor is called immediately to initialize the base-class data members in the derived-class object, then the derived-class constructor initializes the additional derived-class data members.
• When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors—first the derived-class destructor is called, then the base-class destructor is called.

Section 20.6 public, protected and private Inheritance
• Declaring data members private, while providing non-private member functions to manipulate and perform validity checking on this data, enforces good software engineering.
• When deriving a class, the base class may be declared as either public, protected or private.
• When deriving a class from a public base class, public members of the base class become public members of the derived class, and protected members of the base class become protected members of the derived class.
• When deriving a class from a protected base class, public and protected members of the base class become protected members of the derived class.
• When deriving a class from a private base class, public and protected members of the base class become private members of the derived class.

Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>base class</td>
<td>728</td>
</tr>
<tr>
<td>base-class initializer syntax</td>
<td>746</td>
</tr>
<tr>
<td>brittle software</td>
<td>755</td>
</tr>
<tr>
<td>class hierarchy</td>
<td>728</td>
</tr>
<tr>
<td>derived class</td>
<td>728</td>
</tr>
<tr>
<td>direct base class</td>
<td>728</td>
</tr>
<tr>
<td>fragile software</td>
<td>755</td>
</tr>
<tr>
<td>has-a relationship</td>
<td>729</td>
</tr>
<tr>
<td>indirect base class</td>
<td>728</td>
</tr>
<tr>
<td>inheritance</td>
<td>728</td>
</tr>
<tr>
<td>is-a relationship</td>
<td>729</td>
</tr>
<tr>
<td>multiple inheritance</td>
<td>728</td>
</tr>
<tr>
<td>private inheritance</td>
<td>731</td>
</tr>
<tr>
<td>protected inheritance</td>
<td>731</td>
</tr>
<tr>
<td>protected keyword</td>
<td>732</td>
</tr>
<tr>
<td>public inheritance</td>
<td>731</td>
</tr>
<tr>
<td>single inheritance</td>
<td>728</td>
</tr>
<tr>
<td>subclass</td>
<td>728</td>
</tr>
<tr>
<td>superclass</td>
<td>728</td>
</tr>
</tbody>
</table>

Self-Review Exercises

20.1 Fill in the blanks in each of the following statements:

a) ________ is a form of software reuse in which new classes absorb the data and behaviors of existing classes and embellish these classes with new capabilities.
b) A base class’s ________ members can be accessed in the base-class definition, in derived-class definitions and in friends of the base class its derived classes.
c) In a(n) ________ relationship, an object of a derived class also can be treated as an object of its base class.
d) In a(n) _______ relationship, a class object has one or more objects of other classes as members.

e) In single inheritance, a class exists in a(n) _______ relationship with its derived classes.

f) A base class’s _______ members are accessible within that base class and anywhere that the program has a handle to an object of that class or one of its derived classes.

g) A base class’s protected access members have a level of protection between those of public and _______ access.

h) C++ provides for _______, which allows a derived class to inherit from many base classes, even if the base classes are unrelated.

i) When an object of a derived class is instantiated, the base class’s _______ is called implicitly or explicitly to do any necessary initialization of the base-class data members in the derived-class object.

j) When deriving a class from a base class with public inheritance, public members of the base class become _______ members of the derived class, and protected members of the base class become _______ members of the derived class.

k) When deriving a class from a base class with protected inheritance, public members of the base class become _______ members of the derived class, and protected members of the base class become _______ members of the derived class.

20.2 State whether each of the following is true or false. If false, explain why.
a) Base-class constructors are not inherited by derived classes.
b) A has-a relationship is implemented via inheritance.
c) A Car class has an is-a relationship with the SteeringWheel and Brakes classes.
d) Inheritance encourages the reuse of proven high-quality software.
e) When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors.

Answers to Self-Review Exercises

20.1  a) Inheritance.  b) protected.  c) is-a or inheritance.  d) has-a or composition or aggregation.  e) hierarchical.  f) public.  g) private.  h) multiple inheritance.  i) constructor.  j) public, protected.  k) protected, protected.

20.2  a) True.  b) False. A has-a relationship is implemented via composition. An is-a relationship is implemented via inheritance.  c) False. This is an example of a has-a relationship. Class Car has an is-a relationship with class Vehicle.  d) True.  e) True.

Exercises

20.3  (Composition as an Alternative to Inheritance) Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite class BasePlusCommissionEmployee of the CommissionEmployee–BasePlusCommissionEmployee hierarchy to use composition rather than inheritance. After you do this, assess the relative merits of the two approaches for designing classes CommissionEmployee and BasePlusCommissionEmployee, as well as for object-oriented programs in general. Which approach is more natural? Why?

20.4  (Inheritance Advantage) Discuss the ways in which inheritance promotes software reuse, saves time during program development and helps prevent errors.

20.5  (Protected vs. Private Base Classes) Some programmers prefer not to use protected access because they believe it breaks the encapsulation of the base class. Discuss the relative merits of using protected access vs. using private access in base classes.

20.6  (Student Inheritance Hierarchy) Draw an inheritance hierarchy for students at a university similar to the hierarchy shown in Fig. 20.2. Use Student as the base class of the hierarchy, then in-
clude classes UndergraduateStudent and GraduateStudent that derive from Student. Continue to extend the hierarchy as deep (i.e., as many levels) as possible. For example, Freshman, Sophomore, Junior and Senior might derive from UndergraduateStudent, and DoctoralStudent and MastersStudent might derive from GraduateStudent. After drawing the hierarchy, discuss the relationships that exist between the classes. [Note: You do not need to write any code for this exercise.]

20.7 (Richer Shape Hierarchy) The world of shapes is much richer than the shapes included in the inheritance hierarchy of Fig. 20.3. Write down all the shapes you can think of—both two-dimensional and three-dimensional—and form them into a more complete Shape hierarchy with as many levels as possible. Your hierarchy should have the base class Shape from which class TwoDimensionalShape and class ThreeDimensionalShape are derived. [Note: You do not need to write any code for this exercise.] We'll use this hierarchy in the exercises of Chapter 21 to process a set of distinct shapes as objects of base-class Shape. (This technique, called polymorphism, is the subject of Chapter 21.)

20.8 (Quadrilateral Inheritance Hierarchy) Draw an inheritance hierarchy for classes Quadrilateral, Trapezoid, Parallelogram, Rectangle and Square. Use Quadrilateral as the base class of the hierarchy. Make the hierarchy as deep as possible.

20.9 (Package Inheritance Hierarchy) Package-delivery services, such as FedEx®, DHL® and UPS®, offer a number of different shipping options, each with specific costs associated. Create an inheritance hierarchy to represent various types of packages. Use Package as the base class of the hierarchy, then include classes TwoDayPackage and OvernightPackage that derive from Package. Base class Package should include data members representing the name, address, city, state and ZIP code for both the sender and the recipient of the package, in addition to data members that store the weight (in ounces) and cost per ounce to ship the package. Package’s constructor should initialize these data members. Ensure that the weight and cost per ounce contain positive values. Package should provide a public member function calculateCost that returns a double indicating the cost associated with shipping the package. Package’s calculateCost function should determine the cost by multiplying the weight by the cost per ounce. Derived class TwoDayPackage should inherit the functionality of base class Package, but also include a data member that represents a flat fee that the shipping company charges for two-day-delivery service. TwoDayPackage’s constructor should receive a value to initialize this data member. TwoDayPackage should redefine member function calculateCost so that it computes the shipping cost by adding the flat fee to the weight-based cost calculated by base class Package’s calculateCost function. Class OvernightPackage should inherit directly from class Package and contain an additional data member representing an additional fee per ounce charged for overnight-delivery service. OvernightPackage should redefine member function calculateCost so that it adds the additional fee per ounce to the standard cost per ounce before calculating the shipping cost. Write a test program that creates objects of each type of Package and tests member function calculateCost.

20.10 (Account Inheritance Hierarchy) Create an inheritance hierarchy that a bank might use to represent customers’ bank accounts. All customers at this bank can deposit (i.e., credit) money into their accounts and withdraw (i.e., debit) money from their accounts. More specific types of accounts also exist. Savings accounts, for instance, earn interest on the money they hold. Checking accounts, on the other hand, charge a fee per transaction (i.e., credit or debit).

Create an inheritance hierarchy containing base class Account and derived classes SavingsAccount and CheckingAccount that inherit from class Account. Base class Account should include one data member of type double to represent the account balance. The class should provide a constructor that receives an initial balance and uses it to initialize the data member. The constructor should validate the initial balance to ensure that it’s greater than or equal to 0.0. If not, the balance should be set to 0.0 and the constructor should display an error message, indicating that the initial balance was invalid. The class should provide three member functions. Member function credit
Exercises

should add an amount to the current balance. Member function `debit` should withdraw money from the `Account` and ensure that the debit amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the function should print the message "Debit amount exceeded account balance." Member function `getBalance` should return the current balance.

Derived class `SavingsAccount` should inherit the functionality of an `Account`, but also include a data member of type `double` indicating the interest rate (percentage) assigned to the `Account`. `SavingsAccount`'s constructor should receive the initial balance, as well as an initial value for the `SavingsAccount`'s interest rate. `SavingsAccount` should provide a public member function `calculateInterest` that returns a `double` indicating the amount of interest earned by an account. Member function `calculateInterest` should determine this amount by multiplying the interest rate by the account balance. [Note: `SavingsAccount` should inherit member functions `credit` and `debit` as is without redefining them.]

Derived class `CheckingAccount` should inherit from base class `Account` and include an additional data member of type `double` that represents the fee charged per transaction. `CheckingAccount`'s constructor should receive the initial balance, as well as a parameter indicating a fee amount. Class `CheckingAccount` should redefine member functions `credit` and `debit` so that they subtract the fee from the account balance whenever either transaction is performed successfully. `CheckingAccount`'s versions of these functions should invoke the base-class `Account` version to perform the updates to an account balance. `CheckingAccount`'s `debit` function should charge a fee only if money is actually withdrawn (i.e., the debit amount does not exceed the account balance). [Hint: Define `Account`'s `debit` function so that it returns a `bool` indicating whether money was withdrawn. Then use the return value to determine whether a fee should be charged.]

After defining the classes in this hierarchy, write a program that creates objects of each class and tests their member functions. Add interest to the `SavingsAccount` object by first invoking its `calculateInterest` function, then passing the returned interest amount to the object's `credit` function.
Object-Oriented Programming: Polymorphism

One Ring to rule them all, 
One Ring to find them, 
One Ring to bring them all 
and in the darkness bind them.
—John Ronald Reuel Tolkien

The silence often of pure innocence 
Persuades when speaking fails.
—William Shakespeare

General propositions do not decide concrete cases.
—Oliver Wendell Holmes

A philosopher of imposing stature doesn’t think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.
—Alfred North Whitehead

Objectives

In this chapter you’ll learn:

■ How polymorphism makes programming more convenient and systems more extensible.

■ The distinction between abstract and concrete classes and how to create abstract classes.

■ To use runtime type information (RTTI).

■ How C++ implements virtual functions and dynamic binding.

■ How virtual destructors ensure that all appropriate destructors run on an object.
21.1 Introduction

In Chapters 15–20, we discussed key object-oriented programming technologies including classes, objects, encapsulation, operator overloading and inheritance. We now continue our study of OOP by explaining and demonstrating polymorphism with inheritance hierarchies. Polymorphism enables us to “program in the general” rather than “program in the specific.” In particular, polymorphism enables us to write programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy’s base class. As we’ll soon see, polymorphism works off base-class pointer handles and base-class reference handles, but not off name handles.

Suppose we create a polymorphic program that simulates the movement of several types of animals for a biological study. Classes Fish, Frog and Bird represent the three types of animals under investigation. Imagine that each of these classes inherits from base class Animal, which contains a function move and maintains an animal’s current location. Each derived class implements move. Our program maintains a vector of Animal pointers to objects of the derived classes. To simulate the animals’ movements, the program sends each object the same message once per second—namely, move. However, each specific type of Animal responds to a move message in its own way—a Fish might swim two feet, a Frog might jump three feet and a Bird might fly ten feet. The program issues the same message (i.e., move) to each animal object, but each object knows how to modify its location for its specific type of movement. Relying on each object to know how to “do the right thing” in response to the same function call is the key concept of polymorphism. The same message sent to a variety of objects has “many forms” of results—hence the term polymorphism.
With polymorphism, we can design and implement systems that are easily extensible—new classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically. The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that you add to the hierarchy. For example, if we create class `Tortoise` that inherits from class `Animal` (which might respond to a `move` message by crawling one inch), we need to write only the `Tortoise` class and the part of the simulation that instantiates a `Tortoise` object. The portions of the simulation that process each `Animal` generically can remain the same.

We begin with a sequence of small, focused examples that lead up to an understanding of virtual functions and dynamic binding—polymorphism’s two underlying technologies. We then present a case study that revisits Chapter 20’s `Employee` hierarchy. In the case study, we define a common “interface” (i.e., set of functionality) for all the classes in the hierarchy. This common functionality among employees is defined in a so-called abstract base class, `Employee`, from which classes `SalariedEmployee`, `HourlyEmployee` and `CommissionEmployee` inherit directly and class `BaseCommissionEmployee` inherits indirectly. We’ll soon see what makes a class “abstract” or its opposite—“concrete.”

In this hierarchy, every employee has an `earnings` function to calculate the employee’s weekly pay. These `earnings` functions vary by employee type—for instance, `SalariedEmployee`s are paid a fixed weekly salary regardless of the number of hours worked, while `HourlyEmployee`s are paid by the hour and receive overtime pay. We show how to process each employee “in the general”—that is, using base-class pointers to call the `earnings` function of several derived-class objects. This way, you need to be concerned with only one type of function call, which can be used to execute several different functions based on the objects referred to by the base-class pointers.

A key feature of this chapter is its (optional) detailed discussion of polymorphism, virtual functions and dynamic binding “under the hood,” which uses a detailed diagram to explain how polymorphism can be implemented in C++.

Occasionally, when performing polymorphic processing, we need to program “in the specific,” meaning that operations need to be performed on a specific type of object in a hierarchy—the operation cannot be generally applied to several types of objects. We reuse our `Employee` hierarchy to demonstrate the powerful capabilities of runtime type information (RTTI) and dynamic casting, which enable a program to determine the type of an object at execution time and act on that object accordingly. We use these capabilities to determine whether a particular employee object is a `BasePlusCommissionEmployee`, then give that employee a 10 percent bonus on his or her base salary.

### 21.2 Polymorphism Examples

In this section, we discuss several polymorphism examples. With polymorphism, one function can cause different actions to occur, depending on the type of the object on which the function is invoked. This gives you tremendous expressive capability. If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral` object. Therefore, any operation (such as calculating the perimeter or the area) that can be performed on an object of class `Quadrilateral` also can be performed on an object of class `Rectangle`. Such operations also can be performed on other kinds of `Quadrilaterals`, such as `Squares`, `Parallelograms` and `Trapezoids`. The poly-
morphism occurs when a program invokes a virtual function through a base-class (i.e., Quadrilateral) pointer or reference—C++ dynamically (i.e., at execution time) chooses the correct function for the class from which the object was instantiated. You’ll see a code example that illustrates this process in Section 21.3.

As another example, suppose that we design a video game that manipulates objects of many different types, including objects of classes Martian, Venutian, Plutonian, SpaceShip and LaserBeam. Imagine that each of these classes inherits from the common base class SpaceObject, which contains member function draw. Each derived class implements this function in a manner appropriate for that class. A screen-manager program maintains a container (e.g., a vector) that holds SpaceObject pointers to objects of the various classes. To refresh the screen, the screen manager periodically sends each object the same message—namely, draw. Each type of object responds in a unique way. For example, a Martian object might draw itself in red with the appropriate number of antennae. A SpaceShip object might draw itself as a silver flying saucer. A LaserBeam object might draw itself as a bright red beam across the screen. Again, the same message (in this case, draw) sent to a variety of objects has “many forms” of results.

A polymorphic screen manager facilitates adding new classes to a system with minimal modifications to its code. Suppose that we want to add objects of class Mercurian to our video game. To do so, we must build a class Mercurian that inherits from SpaceObject, but provides its own definition of member function draw. Then, when pointers to objects of class Mercurian appear in the container, you do not need to modify the code for the screen manager. The screen manager invokes member function draw on every object in the container, regardless of the object’s type, so the new Mercurian objects simply “plug right in.” Thus, without modifying the system (other than to build and include the classes themselves), you can use polymorphism to accommodate additional classes, including ones that were not even envisioned when the system was created.

Software Engineering Observation 21.1
With virtual functions and polymorphism, you can deal in generalities and let the execution-time environment concern itself with the specifics. You can direct a variety of objects to behave in manners appropriate to those objects without even knowing their types—as long as those objects belong to the same inheritance hierarchy and are being accessed off a common base-class pointer or a common base-class reference.

Software Engineering Observation 21.2
Polymorphism promotes extensibility: Software written to invoke polymorphic behavior is written independently of the types of the objects to which messages are sent. Thus, new types of objects that can respond to existing messages can be incorporated into such a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

21.3 Relationships Among Objects in an Inheritance Hierarchy

Section 20.4 created an employee class hierarchy, in which class BasePlusCommissionEmployee inherited from class CommissionEmployee. The Chapter 20 examples manipulated CommissionEmployee and BasePlusCommissionEmployee objects by using the
Chapter 21 Object-Oriented Programming: Polymorphism

objects’ names to invoke their member functions. We now examine the relationships among classes in a hierarchy more closely. The next several sections present a series of examples that demonstrate how base-class and derived-class pointers can be aimed at base-class and derived-class objects, and how those pointers can be used to invoke member functions that manipulate those objects. In Section 21.3.4, we demonstrate how to get polymorphic behavior from base-class pointers aimed at derived-class objects.

In Section 21.3.1, we assign the address of a derived-class object to a base-class pointer, then show that invoking a function via the base-class pointer invokes the base-class functionality—i.e., the type of the handle determines which function is called. In Section 21.3.2, we assign the address of a base-class object to a derived-class pointer, which results in a compilation error. We discuss the error message and investigate why the compiler does not allow such an assignment. In Section 21.3.3, we assign the address of a derived-class object to a base-class pointer, then examine how the base-class pointer can be used to invoke only the base-class functionality—when we attempt to invoke derived-class member functions through the base-class pointer, compilation errors occur. Finally, in Section 21.3.4, we introduce virtual functions and polymorphism by declaring a base-class function as virtual. We then assign the address of a derived-class object to the base-class pointer and use that pointer to invoke derived-class functionality—precisely the capability we need to achieve polymorphic behavior.

A key concept in these examples is to demonstrate that an object of a derived class can be treated as an object of its base class. This enables various interesting manipulations. For example, a program can create an array of base-class pointers that point to objects of many derived-class types. Despite the fact that the derived-class objects are of different types, the compiler allows this because each derived-class object is an object of its base class. However, we cannot treat a base-class object as an object of any of its derived classes. For example, a CommissionEmployee is not a BasePlusCommissionEmployee in the hierarchy defined in Chapter 20—a CommissionEmployee does not have a baseSalary data member and does not have member functions setBaseSalary and getBaseSalary. The is-a relationship applies only from a derived class to its direct and indirect base classes.

21.3.1 Invoking Base-Class Functions from Derived-Class Objects

The example in Figs. 21.1–21.5 demonstrates three ways to aim base and derived-class pointers at base and derived-class objects. The first two are straightforward—we aim a base-class pointer at a base-class object (and invoke base-class functionality), and we aim a derived-class pointer at a derived-class object (and invoke derived-class functionality). Then, we demonstrate the relationship between derived classes and base classes (i.e., the is-a relationship of inheritance) by aiming a base-class pointer at a derived-class object (and showing that the base-class functionality is indeed available in the derived-class object).

Class CommissionEmployee (Figs. 21.1–21.2), which we discussed in Chapter 20, is used to represent employees who are paid a percentage of their sales. Class BasePlusCommissionEmployee (Figs. 21.3–21.4), which we also discussed in Chapter 20, is used to represent employees who receive a base salary plus a percentage of their sales. Each BasePlusCommissionEmployee object is a CommissionEmployee that also has a base salary. Class BasePlusCommissionEmployee’s earnings member function (lines 30–33 of Fig. 21.4) redefines class CommissionEmployee’s earnings member function (lines 78–81 of Fig. 21.2) to include the object’s base salary. Class BasePlusCommissionEmployee’s
print member function (lines 36–44 of Fig. 21.4) redefines class CommissionEmployee's version (lines 84–91 of Fig. 21.2) to display the same information plus the employee's base salary.

```
// Fig. 21.1: CommissionEmployee.h
// CommissionEmployee class definition represents a commission employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
using namespace std;

class CommissionEmployee {

public:
    CommissionEmployee(const string &, const string &, const string &, double = 0.0, double = 0.0);

    void setFirstName(const string &); // set first name
    string getFirstName() const; // return first name

    void setLastName(const string &); // set last name
    string getLastName() const; // return last name

    void setSocialSecurityNumber(const string &); // set SSN
    string getSocialSecurityNumber() const; // return SSN

    void setGrossSales(double); // set gross sales amount
    double getGrossSales() const; // return gross sales amount

    void setCommissionRate(double); // set commission rate
    double getCommissionRate() const; // return commission rate

    double earnings() const; // calculate earnings
    void print() const; // print CommissionEmployee object

private:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales; // gross weekly sales
    double commissionRate; // commission percentage
};
#endif
```

**Fig. 21.1** | CommissionEmployee class header file.

```
// Fig. 21.2: CommissionEmployee.cpp
// Class CommissionEmployee member-function definitions.
#include <iostream>
#include "CommissionEmployee.h" // CommissionEmployee class definition
```

**Fig. 21.2** | CommissionEmployee class implementation file. (Part 1 of 3.)
Chapter 21 Object-Oriented Programming: Polymorphism

```cpp
using namespace std;

// constructor
CommissionEmployee::CommissionEmployee(
    const string &first, const string &last, const string &ssn,
    double sales, double rate )
    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
{
    setGrossSales( sales ); // validate and store gross sales
    setCommissionRate( rate ); // validate and store commission rate
} // end CommissionEmployee constructor

// set first name
void CommissionEmployee::setFirstName( const string &first )
{
    firstName = first; // should validate
} // end function setFirstName

// return first name
string CommissionEmployee::getFirstName() const
{
    return firstName;
} // end function getFirstName

// set last name
void CommissionEmployee::setLastName( const string &last )
{
    lastName = last; // should validate
} // end function setLastName

// return last name
string CommissionEmployee::getLastName() const
{
    return lastName;
} // end function getLastName

// set social security number
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
    socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string CommissionEmployee::getSocialSecurityNumber() const
{
    return socialSecurityNumber;
} // end function getSocialSecurityNumber

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
} // end function setGrossSales
```

Fig. 21.2 | CommissionEmployee class implementation file. (Part 2 of 3.)
// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
    return grossSales;
} // end function getGrossSales

// set commission rate
void CommissionEmployee::setCommissionRate(double rate)
{
    commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
    return commissionRate;
} // end function getCommissionRate

// calculate earnings
double CommissionEmployee::earnings() const
{
    return getCommissionRate() * getGrossSales();
} // end function earnings

// print CommissionEmployee object
void CommissionEmployee::print() const
{
    cout << "commission employee: "
        << getFirstName() << ' ' << getLastName()
        << "\nsocial security number: " << getSocialSecurityNumber()
        << "\ngross sales: " << getGrossSales()
        << "\ncommission rate: " << getCommissionRate();
} // end function print

Fig. 21.2 | CommissionEmployee class implementation file. (Part 3 of 3.)

// Fig. 21.3: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class
// CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class declaration
using namespace std;

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
    BasePlusCommissionEmployee(const string &, const string &,
                                 const string &, double = 0.0, double = 0.0, double = 0.0);

Fig. 21.3 | BasePlusCommissionEmployee class header file. (Part 1 of 2.)
void setBaseSalary( double ); // set base salary
double getBaseSalary() const; // return base salary
double earnings() const; // calculate earnings
void print() const; // print BasePlusCommissionEmployee object

private:

double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif

// Fig. 21.4: BasePlusCommissionEmployee.cpp
// Class BasePlusCommissionEmployee member-function definitions.
#include <iostream>
#include "BasePlusCommissionEmployee.h"
using namespace std;

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
    const string &first, const string &last, const string &ssn,
    double sales, double rate, double salary )
    : CommissionEmployee( first, last, ssn, sales, rate )
{
    setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
    return baseSalary;
} // end function getBaseSalary

// calculate earnings
double BasePlusCommissionEmployee::earnings() const
{
    return getBaseSalary() + CommissionEmployee::earnings();
} // end function earnings

// print BasePlusCommissionEmployee object
void BasePlusCommissionEmployee::print() const
{
    cout << "base-salaried ";
}
In Fig. 21.5, lines 13–14 create a CommissionEmployee object and line 17 creates a pointer to a CommissionEmployee object; lines 20–21 create a BasePlusCommissionEmployee object and line 24 creates a pointer to a BasePlusCommissionEmployee object. Lines 31 and 33 use each object’s name to invoke its print member function. Line 36 assigns the address of base-class object commissionEmployee to base-class pointer commissionEmployeePtr, which line 39 uses to invoke member function print on that CommissionEmployee object. This invokes the version of print defined in base class CommissionEmployee. Similarly, line 42 assigns the address of derived-class object basePlusCommissionEmployee to derived-class pointer basePlusCommissionEmployeePtr, which line 46 uses to invoke member function print on that BasePlusCommissionEmployee object. This invokes the version of print defined in derived class BasePlusCommissionEmployee. Line 49 then assigns the address of derived-class object basePlusCommissionEmployee to base-class pointer commissionEmployeePtr, which line 53 uses to invoke member function print. This “crossover” is allowed because an object of a derived class is an object of its base class. Note that despite the fact that the base class CommissionEmployee pointer points to a derived class BasePlusCommissionEmployee object, the base class CommissionEmployee’s print member function is invoked (rather than BasePlusCommissionEmployee’s print function). The output of each print member-function invocation in this program reveals that the invoked functionality depends on the type of the handle (i.e., the pointer or reference type) used to invoke the function, not the type of the object to which the handle points. In Section 21.3.4, when we introduce virtual functions, we demonstrate that it’s possible to invoke the object type’s functionality, rather than invoke the handle type’s functionality. We’ll see that this is crucial to implementing polymorphic behavior—the key topic of this chapter.
Chapter 21  Object-Oriented Programming: Polymorphism

// create base-class object
CommissionEmployee commissionEmployee(
    "Sue", "Jones", "222-22-2222", 10000, .06);

// create base-class pointer
CommissionEmployee *commissionEmployeePtr = 0;

// create derived-class object
BasePlusCommissionEmployee basePlusCommissionEmployee(
    "Bob", "Lewis", "333-33-3333", 5000, .04, 300);

// create derived-class pointer
BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

// set floating-point output formatting
cout << fixed << setprecision(2);

// output objects commissionEmployee and basePlusCommissionEmployee
cout << "Print base-class and derived-class objects:

";
commissionEmployee.print(); // invokes base-class print
cout << "

";
basePlusCommissionEmployee.print(); // invokes derived-class print

// aim base-class pointer at base-class object and print
commissionEmployeePtr = &commissionEmployee; // perfectly natural
cout << "\n\nCalling print with base-class pointer to 
" << "\nbase-class object invokes base-class print function:\n\n";
commissionEmployeePtr->print(); // invokes base-class print

// aim derived-class pointer at derived-class object and print
basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
cout << "\n\nCalling print with derived-class pointer to 
" << "\nderived-class object invokes derived-class 
" << "print function:\n\n";
basePlusCommissionEmployeePtr->print(); // invokes derived-class print

// aim base-class pointer at derived-class object and print
commissionEmployeePtr = &basePlusCommissionEmployee;
cout << "\n\nCalling print with base-class pointer to 
" << "derived-class object\ninvokes base-class print 
" << "function on that derived-class object:\n\n";
commissionEmployeePtr->print(); // invokes base-class print

cout << endl;
} // end main

Print base-class and derived-class objects:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Fig. 21.5  Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 2 of 3.)
21.3.2 Aiming Derived-Class Pointers at Base-Class Objects

In Section 21.3.1, we assigned the address of a derived-class object to a base-class pointer and explained that the C++ compiler allows this assignment, because a derived-class object is a base-class object. We take the opposite approach in Fig. 21.6, as we aim a derived-class pointer at a base-class object. [Note: This program uses classes CommissionEmployee and BasePlusCommissionEmployee of Figs. 21.1–21.4.] Lines 8–9 of Fig. 21.6 create a CommissionEmployee object, and line 10 creates a BasePlusCommissionEmployee pointer. Line 14 attempts to assign the address of base-class object commissionEmployee to derived-class pointer basePlusCommissionEmployeePtr, but the C++ compiler generates an error. The compiler prevents this assignment, because a CommissionEmployee is not a BasePlusCommissionEmployee. Consider the consequences if the compiler were to allow this assignment. Through a BasePlusCommissionEmployee pointer, we can invoke every BasePlusCommissionEmployee member function, including setBaseSalary, for the object to which the pointer points (i.e., the base-class object commissionEmployee). However, the CommissionEmployee object does not provide a setBaseSalary member function,
nor does it provide a baseSalary data member to set. This could lead to problems, because member function setBaseSalary would assume that there is a baseSalary data member to set at its “usual location” in a BasePlusCommissionEmployee object. This memory does not belong to the CommissionEmployee object, so member function setBaseSalary might overwrite other important data in memory, possibly data that belongs to a different object.

Fig. 21.7 shows the consequences of attempting to invoke a derived-class member function off a base-class pointer. [Note: We’re again using classes CommissionEmployee and BasePlusCommissionEmployee of Figs. 21.1–21.4.] Line 9 creates commissionEmployeePtr—a pointer to a CommissionEmployee object—and lines 10–11 create a BasePlusCommissionEmployee object. Line 14 aims commissionEmployeePtr at derived-class object basePlusCommissionEmployee. Recall from Section 21.3.1 that this is allowed, because a BasePlusCommissionEmployee is a CommissionEmployee (in the sense that a BasePlusCommissionEmployee object contains all the functionality of a CommissionEmployee object). Lines 18–22 invoke base-class member functions getFirstName, getLastName, getSocialSecurityNumber, getGrossSales and getCommissionRate off the
base-class pointer. All of these calls are legitimate, because `BasePlusCommissionEmployee` inherits these member functions from `CommissionEmployee`. We know that `commissionEmployeePtr` is aimed at a `BasePlusCommissionEmployee` object, so in lines 26–27 we attempt to invoke `BasePlusCommissionEmployee` member functions `getBaseSalary` and `setBaseSalary`. The compiler generates errors on both of these calls, because they’re not made to member functions of base-class `CommissionEmployee`. The handle can be used to invoke only those functions that are members of that handle’s associated class type. (In this case, off a `CommissionEmployee *`, we can invoke only `CommissionEmployee` member functions `setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber`, `getSocialSecurityNumber`, `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` and `print`.)
Chapter 21 Object-Oriented Programming: Polymorphism

The compiler allows access to derived-class-only members from a base-class pointer that is aimed at a derived-class object if we explicitly cast the base-class pointer to a derived-class pointer—known as downcasting. As you know, it's possible to aim a base-class pointer at a derived-class object. However, as we demonstrated in Fig. 21.7, a base-class pointer can be used to invoke only the functions declared in the base class. Downcasting allows a derived-class-specific operation on a derived-class object pointed to by a base-class pointer. After a downcast, the program can invoke derived-class functions that are not in the base class. Section 21.8 shows a concrete example of downcasting.

Fig. 21.7 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 2 of 2.)

The compiler allows access to derived-class-only members from a base-class pointer that is aimed at a derived-class object if we explicitly cast the base-class pointer to a derived-class pointer—known as downcasting. As you know, it's possible to aim a base-class pointer at a derived-class object. However, as we demonstrated in Fig. 21.7, a base-class pointer can be used to invoke only the functions declared in the base class. Downcasting allows a derived-class-specific operation on a derived-class object pointed to by a base-class pointer. After a downcast, the program can invoke derived-class functions that are not in the base class. Section 21.8 shows a concrete example of downcasting.

Software Engineering Observation 21.3

If the address of a derived-class object has been assigned to a pointer of one of its direct or indirect base classes, it's acceptable to cast that base-class pointer back to a pointer of the derived-class type. In fact, this must be done to send that derived-class object messages that do not appear in the base class.

21.3.4 Virtual Functions

In Section 21.3.1, we aimed a base-class CommissionEmployee pointer at a derived-class BasePlusCommissionEmployee object, then invoked member function print through that pointer. Recall that the type of the handle determines which class's functionality to invoke. In that case, the CommissionEmployee pointer invoked the CommissionEmployee member function print on the BasePlusCommissionEmployee object, even though the pointer was aimed at a BasePlusCommissionEmployee object that has its own customized print function. With virtual functions, the type of the object being pointed to, not the type of the handle, determines which version of a virtual function to invoke.

First, we consider why virtual functions are useful. Suppose that shape classes such as Circle, Triangle, Rectangle and Square are all derived from base class Shape. Each of these classes might be endowed with the ability to draw itself via a member function draw. Although each class has its own draw function, the function for each shape is quite different. In a program that draws a set of shapes, it would be useful to be able to treat all the shapes generically as objects of the base class Shape. Then, to draw any shape, we could simply use a base-class Shape pointer to invoke function draw and let the program determine dynamically (i.e., at runtime) which derived-class draw function to use, based on the type of the object to which the base-class Shape pointer points at any given time.

To enable this behavior, we declare draw in the base class as a virtual function, and we override draw in each of the derived classes to draw the appropriate shape. From an implementation perspective, overriding a function is no different than redefining one (which is the approach we've been using until now). An overridden function in a derived class has the same signature and return type (i.e., prototype) as the function it overrides in
its base class. If we do not declare the base-class function as virtual, we can redefine that function. By contrast, if we declare the base-class function as virtual, we can override that function to enable polymorphic behavior. We declare a virtual function by preceding the function’s prototype with the keyword virtual in the base class. For example,

```cpp
virtual void draw() const;
```

would appear in base class Shape. The preceding prototype declares that function `draw` is a virtual function that takes no arguments and returns nothing. This function is declared const because a `draw` function typically would not make changes to the Shape object on which it’s invoked—virtual functions do not have to be const functions.

### Software Engineering Observation 21.4
Once a function is declared virtual, it remains virtual all the way down the inheritance hierarchy from that point, even if that function is not explicitly declared virtual when a derived class overrides it.

### Good Programming Practice 21.1
Even though certain functions are implicitly virtual because of a declaration made higher in the class hierarchy, explicitly declare these functions virtual at every level of the hierarchy to promote program clarity.

### Error-Prevention Tip 21.1
When you browse a class hierarchy to locate a class to reuse, it’s possible that a function in that class will exhibit virtual function behavior even though it isn’t explicitly declared virtual. This happens when the class inherits a virtual function from its base class, and it can lead to subtle logic errors. Such errors can be avoided by explicitly declaring all virtual functions virtual throughout the inheritance hierarchy.

### Software Engineering Observation 21.5
When a derived class chooses not to override a virtual function from its base class, the derived class simply inherits its base class’s virtual function implementation.

If a program invokes a virtual function through a base-class pointer to a derived-class object (e.g., `shapePtr->draw()`) or a base-class reference to a derived-class object (e.g., `shapeRef.draw()`), the program will choose the correct derived-class draw function dynamically (i.e., at execution time) based on the object type—not the pointer or reference type. Choosing the appropriate function to call at execution time (rather than at compile time) is known as dynamic binding or late binding.

When a virtual function is called by referencing a specific object by name and using the dot member-selection operator (e.g., `squareObject.draw()`), the function invocation is resolved at compile time (this is called static binding) and the virtual function that is called is the one defined for (or inherited by) the class of that particular object—this is not polymorphic behavior. Thus, dynamic binding with virtual functions occurs only off pointer (and, as we’ll soon see, reference) handles.

Now let’s see how virtual functions can enable polymorphic behavior in our employee hierarchy. Figures 21.8–21.9 are the header files for classes `CommissionEmployee` and `BasePlusCommissionEmployee`, respectively. The only difference between these files and those of Fig. 21.1 and Fig. 21.3 is that we specify each class’s earnings and
print member functions as virtual (lines 30–31 of Fig. 21.8 and lines 20–21 of Fig. 21.9). Because functions earnings and print are virtual in class CommissionEmployee, class BasePlusCommissionEmployee’s earnings and print functions override class CommissionEmployee’s. Now, if we aim a base-class CommissionEmployee pointer at a derived-class BasePlusCommissionEmployee object, and the program uses that pointer to call either function earnings or print, the BasePlusCommissionEmployee object’s corresponding function will be invoked. There were no changes to the member-function implementations of classes CommissionEmployee and BasePlusCommissionEmployee, so we reuse the versions of Fig. 21.2 and Fig. 21.4.

```cpp
// Fig. 21.8: CommissionEmployee.h
// CommissionEmployee class definition represents a commission employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
using namespace std;

class CommissionEmployee {
public:
    CommissionEmployee( const string &, const string &, const string &,
                        double = 0.0, double = 0.0 );

    void setFirstName( const string & ); // set first name
    string getFirstName() const; // return first name

    void setLastName( const string & ); // set last name
    string getLastName() const; // return last name

    void setSocialSecurityNumber( const string & ); // set SSN
    string getSocialSecurityNumber() const; // return SSN

    void setGrossSales( double ); // set gross sales amount
    double getGrossSales() const; // return gross sales amount

    void setCommissionRate( double ); // set commission rate
    double getCommissionRate() const; // return commission rate

    virtual double earnings() const; // calculate earnings
    virtual void print() const; // print CommissionEmployee object
private:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales; // gross weekly sales
    double commissionRate; // commission percentage
}; // end class CommissionEmployee
#endif
```

**Fig. 21.8** | CommissionEmployee class header file declares earnings and print functions as virtual.
We modified Fig. 21.5 to create the program of Fig. 21.10. Lines 40–51 demonstrate again that a CommissionEmployee pointer aimed at a CommissionEmployee object can be used to invoke CommissionEmployee functionality, and a BasePlusCommissionEmployee pointer aimed at a BasePlusCommissionEmployee object can be used to invoke BasePlusCommissionEmployee functionality. Line 54 aims base-class pointer commissionEmployeePtr at derived-class object basePlusCommissionEmployee. Note that when line 61 invokes member function print off the base-class pointer, the derived-class BasePlusCommissionEmployee's print member function is invoked, so line 61 outputs different text than line 53 does in Fig. 21.5 (when member function print was not declared virtual). We see that declaring a member function virtual causes the program to dynamically determine which function to invoke based on the type of object to which the handle points, rather than on the type of the handle. Note again that when commissionEmployeePtr points to a CommissionEmployee object (line 40), class CommissionEmployee's print function is invoked, and when CommissionEmployeePtr points to a BasePlusCommissionEmployee object, class BasePlusCommissionEmployee's print function is invoked. Thus, the same message—print, in this case—sent (off a base-class pointer) to a variety of objects related by inheritance to that base class, takes on many forms—this is polymorphic behavior.
// Fig. 21.10: fig21_10.cpp
// Introducing polymorphism, virtual functions and dynamic binding.
#include <iostream>
#include <iomanip>
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main()
{
  // create base-class object
  CommissionEmployee commissionEmployee("Sue", "Jones", "222-22-2222", 10000, .06);

  // create base-class pointer
  CommissionEmployee *commissionEmployeePtr = 0;

  // create derived-class object
  BasePlusCommissionEmployee basePlusCommissionEmployee("Bob", "Lewis", "333-33-3333", 5000, .04, 300);

  // create derived-class pointer
  BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

  // set floating-point output formatting
  cout << fixed << setprecision(2);

  // output objects using static binding
  cout << "Invoking print function on base-class and derived-class 
objects with static binding

";
  commissionEmployee.print(); // static binding
  cout << "\n\n";
  basePlusCommissionEmployee.print(); // static binding

  // output objects using dynamic binding
  cout << "\n\nInvoking print function on base-class and \
objects with dynamic binding"

  // aim base-class pointer at base-class object and print
  commissionEmployeePtr = &commissionEmployee;
  cout << "\n\nCalling virtual function print with base-class pointer" \\
<< "\nto base-class object invokes base-class \
function:"
  << "\n"
  commissionEmployeePtr->print(); // invokes base-class print

  // aim derived-class pointer at derived-class object and print
  basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
  cout << "\n\nCalling virtual function print with derived-class " \\
<< "pointer\nto derived-class object invokes derived-class \
function:"
  << "\n"
  basePlusCommissionEmployeePtr->print(); // invokes derived-class print

Fig. 21.10 | Demonstrating polymorphism by invoking a derived-class virtual function via a base-class pointer to a derived-class object. (Part 1 of 2.)
21.3 Relationships Among Objects in an Inheritance Hierarchy

```cpp
// aim base-class pointer at derived-class object and print
commissionEmployeePtr = &basePlusCommissionEmployee;
cout << "\n\nCalling virtual function print with base-class pointer"
<< "\nto derived-class object invokes derived-class "
<< "print function:\n\n";

// polymorphism; invokes BasePlusCommissionEmployee's print;
// base-class pointer to derived-class object
commissionEmployeePtr->print();
cout << endl;
}
} // end main
```

---

**Fig. 21.10** Demonstrating polymorphism by invoking a derived-class virtual function via a base-class pointer to a derived-class object. (Part 2 of 2.)
Chapter 21  Object-Oriented Programming: Polymorphism

21.3.5 Summary of the Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers

Now that you’ve seen a complete application that processes diverse objects polymorphically, we summarize what you can and cannot do with base-class and derived-class objects and pointers. Although a derived-class object also is a base-class object, the two objects are nevertheless different. As discussed previously, derived-class objects can be treated as if they were base-class objects. This is a logical relationship, because the derived class contains all the members of the base class. However, base-class objects cannot be treated as if they were derived-class objects—the derived class can have additional derived-class-only members. For this reason, aiming a derived-class pointer at a base-class object is not allowed without an explicit cast—such an assignment would leave the derived-class-only members undefined on the base-class object. The cast relieves the compiler of the responsibility of issuing an error message. In a sense, by using the cast you’re saying, “I know that what I’m doing is dangerous and I take full responsibility for my actions.”

We’ve discussed four ways to aim base-class pointers and derived-class pointers at base-class objects and derived-class objects:

1. Aiming a base-class pointer at a base-class object is straightforward—calls made off the base-class pointer simply invoke base-class functionality.

2. Aiming a derived-class pointer at a derived-class object is straightforward—calls made off the derived-class pointer simply invoke derived-class functionality.

3. Aiming a base-class pointer at a derived-class object is safe, because the derived-class object is an object of its base class. However, this pointer can be used to invoke only base-class member functions. If you attempt to refer to a derived-class-only member through the base-class pointer, the compiler reports an error. To avoid this error, you must cast the base-class pointer to a derived-class pointer. The derived-class pointer can then be used to invoke the derived-class object’s complete functionality. This technique, called downcasting, is a potentially dangerous operation—Section 21.8 demonstrates how to safely use downcasting. If a virtual function is defined in the base and derived classes (either by inheritance or overriding), and if that function is invoked on a derived-class object via a base-class pointer, then the derived-class version of that function is called. This is an example of the polymorphic behavior that occurs only with virtual functions.

4. Aiming a derived-class pointer at a base-class object generates a compilation error. The is-a relationship applies only from a derived class to its direct and indirect base classes, and not vice versa. A base-class object does not contain the derived-class-only members that can be invoked off a derived-class pointer.

---

**Common Programming Error 21.1**

After aiming a base-class pointer at a derived-class object, attempting to reference derived-class-only members with the base-class pointer is a compilation error.

**Common Programming Error 21.2**

Treating a base-class object as a derived-class object can cause errors.
21.4 Type Fields and switch Statements

One way to determine the type of an object is to use a switch statement to check the value of a field in the object. This allows us to distinguish among object types, then invoke an appropriate action for a particular object. For example, in a hierarchy of shapes in which each shape object has a shapeType attribute, a switch statement could check the object’s shapeType to determine which print function to call.

Using switch logic exposes programs to a variety of potential problems. For example, you might forget to include a type test when one is warranted, or might forget to test all possible cases in a switch statement. When modifying a switch-based system by adding new types, you might forget to insert the new cases in all relevant switch statements. Every addition or deletion of a class requires the modification of every switch statement in the system; tracking these statements down can be time consuming and error prone.

Software Engineering Observation 21.6
Polymorphic programming can eliminate the need for switch logic. By using the polymorphism mechanism to perform the equivalent logic, you can avoid the kinds of errors typically associated with switch logic.

Software Engineering Observation 21.7
An interesting consequence of using polymorphism is that programs take on a simplified appearance. They contain less branching logic and simpler sequential code. This simplification facilitates testing, debugging and program maintenance.

21.5 Abstract Classes and Pure virtual Functions

When we think of a class as a type, we assume that programs will create objects of that type. However, there are cases in which it’s useful to define classes from which you never intend to instantiate any objects. Such classes are called abstract classes. Because these classes normally are used as base classes in inheritance hierarchies, we refer to them as abstract base classes. These classes cannot be used to instantiate objects, because, as we’ll soon see, abstract classes are incomplete—derived classes must define the “missing pieces.”

We build programs with abstract classes in Section 21.6.

An abstract class provides a base class from which other classes can inherit. Classes that can be used to instantiate objects are called concrete classes. Such classes define every member function they declare. We could have an abstract base class TwoDimensionalShape and derive such concrete classes as Square, Circle and Triangle. We could also have an abstract base class ThreeDimensionalShape and derive such concrete classes as Cube, Sphere and Cylinder. Abstract base classes are too generic to define real objects; we need to be more specific before we can think of instantiating objects. For example, if someone tells you to “draw the two-dimensional shape,” what shape would you draw? Concrete classes provide the specifics that make it reasonable to instantiate objects.

An inheritance hierarchy does not need to contain any abstract classes, but many object-oriented systems have class hierarchies headed by abstract base classes. In some cases, abstract classes constitute the top few levels of the hierarchy. A good example of this is the shape hierarchy in Fig. 20.3, which begins with abstract base class Shape. On the next level of the hierarchy we have two more abstract base classes, namely, TwoDimensional-
alShape and ThreeDimensionalShape. The next level of the hierarchy defines concrete classes for two-dimensional shapes (namely, Circle, Square and Triangle) and for three-dimensional shapes (namely, Sphere, Cube and Tetrahedron).

A class is made abstract by declaring one or more of its virtual functions to be “pure.” A pure virtual function is specified by placing “= 0” in its declaration, as in

```cpp
virtual void draw() const = 0; // pure virtual function
```

The “= 0” is a pure specifier. Pure virtual functions do not provide implementations. Every concrete derived class must override all base-class pure virtual functions with concrete implementations of those functions. The difference between a virtual function and a pure virtual function is that a virtual function has an implementation and gives the derived class the option of overriding the function; by contrast, a pure virtual function does not provide an implementation and requires the derived class to override the function for that derived class to be concrete; otherwise the derived class remains abstract.

Pure virtual functions are used when it does not make sense for the base class to have an implementation of a function, but you want all concrete derived classes to implement the function. Returning to our earlier example of space objects, it does not make sense for the base class SpaceObject to have an implementation for function draw (as there is no way to draw a generic space object without having more information about what type of space object is being drawn). An example of a function that would be defined as virtual (and not pure virtual) would be one that returns a name for the object. We can name a generic SpaceObject (for instance, as "space object"), so a default implementation for this function can be provided, and the function does not need to be pure virtual. The function is still declared virtual, however, because it’s expected that derived classes will override this function to provide more specific names for the derived-class objects.

**Software Engineering Observation 21.8**

An abstract class defines a common public interface for the various classes in a class hierarchy. An abstract class contains one or more pure virtual functions that concrete derived classes must override.

**Common Programming Error 21.3**

Attempting to instantiate an object of an abstract class causes a compilation error.

**Common Programming Error 21.4**

Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

**Software Engineering Observation 21.9**

An abstract class has at least one pure virtual function. An abstract class also can have data members and concrete functions (including constructors and destructors), which are subject to the normal rules of inheritance by derived classes.

Although we cannot instantiate objects of an abstract base class, we can use the abstract base class to declare pointers and references that can refer to objects of any concrete classes derived from the abstract class. Programs typically use such pointers and references to manipulate derived-class objects polymorphically.
Consider another application of polymorphism. A screen manager needs to display a variety of objects, including new types of objects that you’ll add to the system after writing the screen manager. The system might need to display various shapes, such as Circles, Triangles or Rectangles, which are derived from abstract base class Shape. The screen manager uses Shape pointers to manage the objects that are displayed. To draw any object (regardless of the level at which that object’s class appears in the inheritance hierarchy), the screen manager uses a base-class pointer to the object to invoke the object’s draw function, which is a pure virtual function in base class Shape; therefore, each concrete derived class must implement function draw. Each Shape object in the inheritance hierarchy knows how to draw itself. The screen manager does not have to worry about the type of each object or whether the screen manager has ever encountered objects of that type.

Polymorphism is particularly effective for implementing layered software systems. In operating systems, for example, each type of physical device could operate quite differently from the others. Even so, commands to read or write data from and to devices may have a certain uniformity. The write message sent to a device-driver object needs to be interpreted specifically in the context of that device driver and how that device driver manipulates devices of a specific type. However, the write call itself really is no different from the write to any other device in the system—place some number of bytes from memory onto that device. An object-oriented operating system might use an abstract base class to provide an interface appropriate for all device drivers. Then, through inheritance from that abstract base class, derived classes are formed that all operate similarly. The capabilities (i.e., the public functions) offered by the device drivers are provided as pure virtual functions in the abstract base class. The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of device drivers. This architecture also allows new devices to be added to a system easily, even after the operating system has been defined. The user can just plug in the device and install its new device driver. The operating system “talks” to this new device through its device driver, which has the same public member functions as all other device drivers—those defined in the device driver abstract base class.

It’s common in object-oriented programming to define an iterator class that can traverse all the objects in a container (such as an array). For example, a program can print a list of objects in a vector by creating an iterator object, then using the iterator to obtain the next element of the list each time the iterator is called. Iterators often are used in polymorphic programming to traverse an array or a linked list of pointers to objects from various levels of a hierarchy. The pointers in such a list are all base-class pointers. A list of pointers to objects of base class TwoDimensionalShape could contain pointers to objects of classes Square, Circle, Triangle and so on. Using polymorphism to send a draw message, off a TwoDimensionalShape * pointer, to each object in the list would draw each object correctly on the screen.

## 21.6 Case Study: Payroll System Using Polymorphism

This section reexamines the CommissionEmployee-BasePlusCommissionEmployee hierarchy that we explored throughout Section 20.4. In this example, we use an abstract class and polymorphism to perform payroll calculations based on the type of employee. We create an enhanced employee hierarchy to solve the following problem:
A company pays its employees weekly. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales and base-salary-plus-commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward base-salary-plus-commission employees by adding 10 percent to their base salaries. The company wants to implement a C++ program that performs its payroll calculations polymorphically.

We use abstract class Employee to represent the general concept of an employee. The classes that derive directly from Employee are SalariedEmployee, CommissionEmployee and HourlyEmployee. Class BasePlusCommissionEmployee—derived from CommissionEmployee—represents the last employee type. The UML class diagram in Fig. 21.11 shows the inheritance hierarchy for our polymorphic employee payroll application. The abstract class name Employee is italicized, as per the convention of the UML.

Abstract base class Employee declares the “interface” to the hierarchy—that is, the set of member functions that a program can invoke on all Employee objects. Each employee, regardless of the way his or her earnings are calculated, has a first name, a last name and a social security number, so private data members firstName, lastName and socialSecurityNumber appear in abstract base class Employee.

**Software Engineering Observation 21.10**

A derived class can inherit interface or implementation from a base class. Hierarchies designed for implementation inheritance tend to have their functionality high in the hierarchy—each new derived class inherits one or more member functions that were defined in a base class, and the derived class uses the base-class definitions. Hierarchies designed for interface inheritance tend to have their functionality lower in the hierarchy—a base class specifies one or more functions that should be defined for each class in the hierarchy (i.e., they have the same prototype), but the individual derived classes provide their own implementations of the function(s).

The following sections implement the Employee class hierarchy. The first five each implement one of the abstract or concrete classes. The last section implements a test program that builds objects of all these classes and processes the objects polymorphically.
21.6.1 Creating Abstract Base Class Employee

Class Employee (Figs. 21.13–21.14, discussed in further detail shortly) provides functions earnings and print, in addition to various get and set functions that manipulate Employee’s data members. An earnings function certainly applies generically to all employees, but each earnings calculation depends on the employee’s class. So we declare earnings as pure virtual in base class Employee because a default implementation does not make sense for that function—there is not enough information to determine what amount earnings should return. Each derived class overrides earnings with an appropriate implementation. To calculate an employee’s earnings, the program assigns the address of an employee’s object to a base class Employee pointer, then invokes the earnings function on that object. We maintain a vector of Employee pointers, each of which points to an Employee object (of course, there cannot be Employee objects, because Employee is an abstract class—because of inheritance, however, all objects of all derived classes of Employee may nevertheless be thought of as Employee objects). The program iterates through the vector and calls function earnings for each Employee object. C++ processes these function calls polymorphically. Including earnings as a pure virtual function in Employee forces every direct derived class of Employee that wishes to be a concrete class to override earnings. This enables the designer of the class hierarchy to demand that each derived class provide an appropriate pay calculation, if indeed that derived class is to be concrete.

Function print in class Employee displays the first name, last name and social security number of the employee. As we’ll see, each derived class of Employee overrides function print to output the employee’s type (e.g., “salaried employee:”) followed by the rest of the employee’s information. Function print could also call earnings, even though earnings is a pure virtual function in class Employee.

The diagram in Fig. 21.12 shows each of the five classes in the hierarchy down the left side and functions earnings and print across the top. For each class, the diagram shows the desired results of each function. Class Employee specifies “= 0” for function earnings to indicate that this is a pure virtual function. Each derived class overrides this function to provide an appropriate implementation. We do not list base class Employee’s get and set functions because they’re not overridden in any of the derived classes—each of these functions is inherited and used “as is” by each of the derived classes.

Let’s consider class Employee’s header file (Fig. 21.13). The public member functions include a constructor that takes the first name, last name and social security number as arguments (line 12); set functions that set the first name, last name and social security number (lines 14, 17 and 20, respectively); get functions that return the first name, last name and social security number (lines 15, 18 and 21, respectively); pure virtual function earnings (line 24) and virtual function print (line 25).

Recall that we declared earnings as a pure virtual function because first we must know the specific Employee type to determine the appropriate earnings calculations. Declaring this function as pure virtual indicates that each concrete derived class must provide an earnings implementation and that a program can use base-class Employee pointers to invoke function earnings polymorphically for any type of Employee.

Figure 21.14 contains the member-function implementations for class Employee. No implementation is provided for virtual function earnings. The Employee constructor (lines 9–14) does not validate the social security number. Normally, such validation should be provided.
```cpp
// Fig. 21.13: Employee.h
// Employee abstract base class.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string> // C++ standard string class
using namespace std;

class Employee
{
public:
    Employee( const string &, const string &, const string & );

    void setFirstName( const string & ); // set first name
    string getFirstName() const; // return first name

    void setLastName( const string & ); // set last name
    string getLastName() const; // return last name

    void setSocialSecurityNumber( const string & ); // set SSN
    string getSocialSecurityNumber() const; // return SSN
};
```

**Fig. 21.13 |** Employee class header file. (Part 1 of 2.)
// pure virtual function makes Employee abstract base class
virtual double earnings() const = 0; // pure virtual
virtual void print() const; // virtual

private:
    string firstName;
    string lastName;
    string socialSecurityNumber;
}; // end class Employee

#endif // EMPLOYEE_H

Fig. 21.13 | Employee class header file. (Part 2 of 2.)

// Fig. 21.14: Employee.cpp
// Abstract-base-class Employee member-function definitions.
// Note: No definitions are given for pure virtual functions.
#include <iostream>
#include "Employee.h" // Employee class definition
using namespace std;

// constructor
Employee::Employee( const string &first, const string &last, const string &ssn )
    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
{ // empty body
} // end Employee constructor

// set first name
void Employee::setFirstName( const string &first )
{ 
    firstName = first;
} // end function setFirstName

// return first name
string Employee::getFirstName() const
{ 
    return firstName;
} // end function getFirstName

// set last name
void Employee::setLastName( const string &last )
{ 
    lastName = last;
} // end function setLastName

// return last name
string Employee::getLastName() const
{ 
    return lastName;
} // end function getLastName

Fig. 21.14 | Employee class implementation file. (Part 1 of 2.)
Note that virtual function print (Fig. 21.14, lines 53–57) provides an implementation that will be overridden in each of the derived classes. Each of these functions will, however, use the abstract class’s version of print to print information common to all classes in the Employee hierarchy.

### 21.6.2 Creating Concrete Derived Class SalariedEmployee

Class SalariedEmployee (Figs. 21.15–21.16) derives from class Employee (line 8 of Fig. 21.15). The public member functions include a constructor that takes a first name, a last name, a social security number and a weekly salary as arguments (lines 11–12); a set function to assign a new nonnegative value to data member weeklySalary (line 14); a get function to return weeklySalary’s value (line 15); a virtual function earnings that calculates a SalariedEmployee’s earnings (line 18) and a virtual function print (line 19) that outputs the employee’s type, namely, "salaried employee: " followed by employee-specific information produced by base class Employee’s print function and SalariedEmployee’s getWeeklySalary function.
Case Study: Payroll System Using Polymorphism

Figure 21.16 contains the member-function implementations for SalariedEmployee. The class’s constructor passes the first name, last name and social security number to the Employee constructor (line 10) to initialize the private data members that are inherited from the base class, but not accessible in the derived class. Function earnings (line 29–32) overrides pure virtual function earnings in Employee to provide a concrete implementation that returns the SalariedEmployee’s weekly salary. If we did not implement earnings, class SalariedEmployee would be an abstract class, and any attempt to instantiate an object of the class would result in a compilation error (and, of course, we want SalariedEmployee here to be a concrete class). In class SalariedEmployee’s header file, we declared member functions earnings and print as virtual (lines 18–19 of Fig. 21.15)—actually, placing the virtual keyword before these member functions is redundant. We defined them as virtual in base class Employee, so they remain virtual functions throughout the class hierarchy. Recall from Good Programming Practice 21.1 that explicitly declaring such functions virtual at every level of the hierarchy can promote program clarity.
Function print of class SalariedEmployee (lines 35–40 of Fig. 21.16) overrides Employee function print. If class SalariedEmployee did not override print, SalariedEmployee would inherit the Employee version of print. In that case, SalariedEmployee's print function would simply return the employee's full name and social security number, which does not adequately represent a SalariedEmployee. To print a SalariedEmployee's complete information, the derived class's print function outputs "salaried employee: " followed by the base-class Employee-specific information (i.e., first name, last name and social security number) printed by invoking the base class's print function using the scope resolution operator (line 38)—this is a nice example of code reuse. The output produced by SalariedEmployee's print function contains the employee's weekly salary obtained by invoking the class's getWeeklySalary function.

### 21.6.3 Creating Concrete Derived Class HourlyEmployee

Class HourlyEmployee (Figs. 21.17–21.18) also derives from class Employee (line 8 of Fig. 21.17). The public member functions include a constructor (lines 13–14) that takes as arguments a first name, a last name, a social security number, an hourly wage and the number of hours worked; set functions that assign new values to data members wage and hours, respectively (lines 16 and 19); get functions to return the values of wage and hours,
respectively (lines 17 and 20); a virtual function earnings that calculates an HourlyEmployee's earnings (line 23) and a virtual function print that outputs the employee's type, namely, "hourly employee: " and employee-specific information (line 24).

Figure 21.17 contains the member-function implementations for class HourlyEmployee. Lines 17–20 and 29–33 define set functions that assign new values to data members wage and hours, respectively. Function setWage (lines 17–20) ensures that wage is non-negative, and function setHours (lines 29–33) ensures that data member hours is between 0 and hoursPerWeek (i.e., 168). Class HourlyEmployee's get functions are implemented in lines 23–26 and 36–39. We do not declare these functions virtual, so classes derived from class HourlyEmployee cannot override them (although derived classes certainly can redefine them). The HourlyEmployee constructor, like the SalariedEmployee constructor, passes the first name, last name and social security number to the base class Employee constructor (line 10) to initialize the inherited private data members declared in the base class. In addition, HourlyEmployee's print function calls base-class function print (line 55) to output the Employee-specific information (i.e., first name, last name and social security number)—this is another nice example of code reuse.
// Fig. 21.18: HourlyEmployee.cpp
// HourlyEmployee class member-function definitions.
#include <iostream>
#include "HourlyEmployee.h" // HourlyEmployee class definition
using namespace std;

// constructor
HourlyEmployee::HourlyEmployee( const string &first, const string &last, const string &ssn, double hourlyWage, double hoursWorked ) :
   Employee( first, last, ssn )
{
   setWage( hourlyWage ); // validate hourly wage
   setHours( hoursWorked ); // validate hours worked
} // end HourlyEmployee constructor

// set wage
void HourlyEmployee::setWage( double hourlyWage )
{
   wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
} // end function setWage

// return wage
double HourlyEmployee::getWage() const
{
   return wage;
} // end function getWage

// set hours worked
void HourlyEmployee::setHours( double hoursWorked )
{
   hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= hoursPerWeek ) ) ? hoursWorked : 0.0 ;
} // end function setHours

// return hours worked
double HourlyEmployee::getHours() const
{
   return hours;
} // end function getHours

// calculate earnings;
// override pure virtual function earnings in Employee
double HourlyEmployee::earnings() const
{
   if ( getHours() <= 40 ) // no overtime
      return getWage() * getHours();
   else
      return 40 * getWage() + ( ( getHours() - 40 ) * getWage() * 1.5 );
} // end function earnings

// print HourlyEmployee's information
void HourlyEmployee::print() const
21.6.4 Creating Concrete Derived Class CommissionEmployee

Class CommissionEmployee (Figs. 21.19–21.20) derives from Employee (Fig. 21.19, line 8). The member-function implementations (Fig. 21.20) include a constructor (lines 8–14) that takes a first name, last name, social security number, sales amount and commission rate; set functions (lines 17–20 and 29–32) to assign new values to data members commissionRate and grossSales, respectively; get functions (lines 23–26 and 35–38) that retrieve their values; function earnings (lines 41–44) to calculate a CommissionEmployee's earnings; and function print (lines 47–53) to output the employee's type, namely, "commission employee: " and employee-specific information. The constructor passes the first name, last name and social security number to the Employee constructor (line 10) to initialize Employee's private data members. Function print calls base-class function print (line 50) to display the Employee-specific information.
// Fig. 21.20: CommissionEmployee.cpp
// CommissionEmployee class member-function definitions.
#include <iostream>
#include "CommissionEmployee.h" // CommissionEmployee class definition
using namespace std;

// constructor
CommissionEmployee::CommissionEmployee( const string &first,
const string &last, const string &ssn, double sales, double rate ) :
Employee( first, last, ssn )
{
    setGrossSales( sales );
    setCommissionRate( rate );
} // end CommissionEmployee constructor

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
    commissionRate = ( ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0 );
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
    return commissionRate;
} // end function getCommissionRate

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
    grossSales = ( ( sales < 0.0 ) ? 0.0 : sales );
} // end function setGrossSales

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
    return grossSales;
} // end function getGrossSales

// calculate earnings; override pure virtual function earnings in Employee
double CommissionEmployee::earnings() const
{
    return getCommissionRate() * getGrossSales();
} // end function earnings

// print CommissionEmployee's information
void CommissionEmployee::print() const
{
    cout << "commission employee: ";
    Employee::print(); // code reuse
    cout << "gross sales: " << getGrossSales() << "; commission rate: " << getCommissionRate();
} // end function print

Fig. 21.20 | CommissionEmployee class implementation file.
21.6.5 Creating Indirect Concrete Derived Class

Class `BasePlusCommissionEmployee` (Figs. 21.21–21.22) directly inherits from class `CommissionEmployee` (line 8 of Fig. 21.21) and therefore is an indirect derived class of class `Employee`. Class `BasePlusCommissionEmployee`'s member-function implementations include a constructor (lines 8–14 of Fig. 21.22) that takes as arguments a first name, a last name, a social security number, a sales amount, a commission rate and a base salary. It then passes the first name, last name, social security number, sales amount and commission rate to the `CommissionEmployee` constructor (line 11) to initialize the inherited members. `BasePlusCommissionEmployee` also contains a set function (lines 17–20) to assign a new value to data member `baseSalary` and a get function (lines 23–26) to return `baseSalary`'s value. Function `earnings` (lines 30–33) calculates a `BasePlusCommissionEmployee`'s earnings. Line 32 in function `earnings` calls base-class `CommissionEmployee`'s `earnings` function to calculate the commission-based portion of the employee's earnings. This is a nice example of code reuse. `BasePlusCommissionEmployee`'s `print` function (lines 36–41) outputs "base-salaried", followed by the output of base-class `CommissionEmployee`'s `print` function (another example of code reuse), then the base salary. The resulting output begins with "base-salaried commission employee: " followed by the rest of the `BasePlusCommissionEmployee`'s information. Recall that `CommissionEmployee`'s `print` displays the employee's first name, last name and social security number by invoking the `print` function of its base class (i.e., `Employee`)—yet another example of code reuse. `BasePlusCommissionEmployee`'s `print` initiates a chain of functions calls that spans all three levels of the `Employee` hierarchy.

```cpp
// Fig. 21.21: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include "CommissionEmployee.h" // CommissionEmployee class definition

class BasePlusCommissionEmployee : public CommissionEmployee
{

  public:

    BasePlusCommissionEmployee( const string &, const string &,
                                 const string &, double = 0.0, double = 0.0, double = 0.0 );

    void setBaseSalary( double ); // set base salary
    double getBaseSalary() const; // return base salary

    // keyword virtual signals intent to override
    virtual double earnings() const; // calculate earnings
    virtual void print() const; // print BasePlusCommissionEmployee object

  private:
    double baseSalary; // base salary per week
}; // end class BasePlusCommissionEmployee

#endif // BASEPLUS_H
```

**Fig. 21.21** BasePlusCommissionEmployee class header file.
21.6.6 Demonstrating Polymorphic Processing

To test our Employee hierarchy, the program in Fig. 21.23 creates an object of each of the four concrete classes SalariedEmployee, HourlyEmployee, CommissionEmployee and BasePlusCommissionEmployee. The program manipulates these objects, first with static binding, then polymorphically, using a vector of Employee pointers. Lines 23–30 create objects of each of the four concrete Employee derived classes. Lines 35–43 output each Employee’s information and earnings. Each member-function invocation in lines 35–43 is an example of static binding—at compile time, because we’re using name handles (not
// Fig. 21.23: fig21_23.cpp
// Processing Employee derived-class objects individually
// and polymorphically using dynamic binding.
#include <iostream>
#include <iomanip>
#include <vector>
#include "Employee.h"
#include "SalariedEmployee.h"
#include "HourlyEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

void virtualViaPointer(const Employee * const); // prototype
void virtualViaReference(const Employee &); // prototype

int main()
{
    // set floating-point output formatting
    cout << fixed << setprecision(2);

    // create derived-class objects
    SalariedEmployee salariedEmployee(
        "John", "Smith", "111-11-1111", 800);
    HourlyEmployee hourlyEmployee(
        "Karen", "Price", "222-22-2222", 16.75, 40);
    CommissionEmployee commissionEmployee(
        "Sue", "Jones", "333-33-3333", 10000, .06);
    BasePlusCommissionEmployee basePlusCommissionEmployee(
        "Bob", "Lewis", "444-44-4444", 5000, .04, 300);

    cout << "Employees processed individually using static binding:

";

    // output each Employee's information and earnings using static binding
    salariedEmployee.print();
    cout << "earned $" << salariedEmployee.earnings() << "\n\n";
    hourlyEmployee.print();
    cout << "earned $" << hourlyEmployee.earnings() << "\n\n";
    commissionEmployee.print();
    cout << "earned $" << commissionEmployee.earnings() << "\n\n";
    basePlusCommissionEmployee.print();
    cout << "earned $" << basePlusCommissionEmployee.earnings() << "\n\n";

    // create vector of four base-class pointers
    vector < Employee * > employees(4);

    // initialize vector with Employees
    employees[0] = &salariedEmployee;
    employees[1] = &hourlyEmployee;
    employees[2] = &commissionEmployee;
}
Chapter 21  Object-Oriented Programming: Polymorphism

```
50  employees[ 1 ] = &hourlyEmployee;
51  employees[ 2 ] = &commissionEmployee;
53  
54  cout << "Employees processed polymorphically via dynamic binding:\n\n";
55  // call virtualViaPointer to print each Employee's information
56  // and earnings using dynamic binding
57  cout << "Virtual function calls made off base-class pointers:\n\n";
58  
59  for ( size_t i = 0; i < employees.size(); i++ )
60      virtualViaPointer( employees[ i ] );
61  
62  // call virtualViaReference to print each Employee's information
63  // and earnings using dynamic binding
64  cout << "Virtual function calls made off base-class references:\n\n";
65  
66  for ( size_t i = 0; i < employees.size(); i++ )
67      virtualViaReference( *employees[ i ] ); // note dereferencing
68  
69  // call Employee virtual functions print and earnings off a
70  // base-class pointer using dynamic binding
71  void virtualViaPointer( const Employee * const baseClassPtr )
72  {
73     baseClassPtr->print();
74     cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
75  } // end function virtualViaPointer
76  
77  // call Employee virtual functions print and earnings off a
78  // base-class reference using dynamic binding
79  void virtualViaReference( const Employee &baseClassRef )
80  {
81     baseClassRef.print();
82     cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
83  } // end function virtualViaReference
```

Employees processed individually using static binding:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

Fig. 21.23  |  Employee class hierarchy driver program. (Part 2 of 3.)
assignments, because a SalariedEmployee is an Employee, an HourlyEmployee is an Employee, a CommissionEmployee is an Employee and a BasePlusCommissionEmployee is an Employee. Therefore, we can assign the addresses of SalariedEmployee, HourlyEmployee, CommissionEmployee and BasePlusCommissionEmployee objects to base-class Employee pointers (even though Employee is an abstract class).

The loop in lines 60–61 traverses vector employees and invokes function virtualViaPointer (lines 73–77) for each element in employees. Function virtualViaPointer receives in parameter baseClassPtr (of type const Employee * const) the address stored in an employees element. Each call to virtualViaPointer uses baseClassPtr to invoke virtual functions print (line 75) and earnings (line 76). Note that function virtualViaPointer does not contain any SalariedEmployee, HourlyEmployee, CommissionEmployee or BasePlusCommissionEmployee type information. The function knows only about base-class type Employee. Therefore, the compiler cannot know which concrete class’s functions to call through baseClassPtr. Yet at execution time, each virtual-function invocation calls the function on the object to which baseClassPtr points at that moment. The output illustrates that the appropriate functions for each class are indeed invoked and that each object’s proper information is displayed. For instance, the weekly salary is displayed for the SalariedEmployee, and the gross sales are displayed for the CommissionEmployee and BasePlusCommissionEmployee. Also, obtaining the earnings of each Employee polymorphically in line 76 produces the same results as obtaining these employees’ earnings via static binding in lines 36, 38, 40 and 42. All virtual function calls to print and earnings are resolved at runtime with dynamic binding.

Finally, another for statement (lines 67–68) traverses employees and invokes function virtualViaReference (lines 81–85) for each element in the vector. Function virtualViaReference receives in its parameter baseClassRef (of type const Employee &) a reference to the object obtained by dereferencing the pointer stored in each employees element (line 68). Each call to virtualViaReference invokes virtual functions print (line 83) and earnings (line 84) via reference baseClassRef to demonstrate that polymorphic processing occurs with base-class references as well. Each virtual-function invocation calls the function on the object to which baseClassRef refers at runtime. This is another example of dynamic binding. The output produced using base-class references is identical to the output produced using base-class pointers.

21.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”

C++ makes polymorphism easy to program. It’s certainly possible to program for polymorphism in non-object-oriented languages such as C, but doing so requires complex and potentially dangerous pointer manipulations. This section discusses how C++ can implement polymorphism, virtual functions and dynamic binding internally. This will give you a solid understanding of how these capabilities really work. More importantly, it will help you appreciate the overhead of polymorphism—in terms of additional memory consumption and processor time. This will help you determine when to use polymorphism and when to avoid it. The STL components were implemented without polymorphism and virtual functions—this was done to avoid the associated execution-time overhead and achieve optimal performance to meet the unique requirements of the STL.
First, we’ll explain the data structures that the C++ compiler builds at compile time to support polymorphism at execution time. You’ll see that polymorphism is accomplished through three levels of pointers (i.e., “triple indirection”). Then we’ll show how an executing program uses these data structures to execute virtual functions and achieve the dynamic binding associated with polymorphism. Our discussion explains one possible implementation; this is not a language requirement.

When C++ compiles a class that has one or more virtual functions, it builds a virtual function table (vtable) for that class. An executing program uses the vtable to select the proper function implementation each time a virtual function of that class is called. The leftmost column of Fig. 21.24 illustrates the vtables for classes Employee, SalariedEmployee, HourlyEmployee, CommissionEmployee and BasePlusCommissionEmployee.

In the vtable for class Employee, the first function pointer is set to 0 (i.e., the null pointer). This is done because function earnings is a pure virtual function and therefore lacks an implementation. The second function pointer points to function print, which displays the employee’s full name and social security number. [Note: We’ve abbreviated the output of each print function in this figure to conserve space.] Any class that has one or more null pointers in its vtable is an abstract class. Classes without any null vtable pointers (such as SalariedEmployee, HourlyEmployee, CommissionEmployee and BasePlusCommissionEmployee) are concrete classes.

Class SalariedEmployee overrides function earnings to return the employee’s weekly salary, so the function pointer points to the earnings function of class SalariedEmployee. SalariedEmployee also overrides print, so the corresponding function pointer points to the SalariedEmployee member function that prints "salaried employee: " followed by the employee’s name, social security number and weekly salary.

The earnings function pointer in the vtable for class HourlyEmployee points to HourlyEmployee’s earnings function that returns the employee’s wage multiplied by the number of hours worked. To conserve space, we’ve omitted the fact that hourly employees receive time-and-a-half pay for overtime hours worked. The print function pointer points to the HourlyEmployee version of the function, which prints "hourly employee: ", the employee’s name, social security number, hourly wage and hours worked. Both functions override the functions in class Employee.

The earnings function pointer in the vtable for class CommissionEmployee points to CommissionEmployee’s earnings function that returns the employee’s gross sales multiplied by the commission rate. The print function pointer points to the CommissionEmployee version of the function, which prints the employee’s type, name, social security number, commission rate and gross sales. As in class HourlyEmployee, both functions override the functions in class Employee.

The earnings function pointer in the vtable for class BasePlusCommissionEmployee points to the BasePlusCommissionEmployee’s earnings function, which returns the employee’s base salary plus gross sales multiplied by commission rate. The print function pointer points to the BasePlusCommissionEmployee version of the function, which prints the employee’s base salary plus the type, name, social security number, commission rate and gross sales. Both functions override the functions in class CommissionEmployee.

Notice that in our Employee case study, each concrete class provides its own implementation for virtual functions earnings and print. You’ve learned that each class which inherits directly from abstract base class Employee must implement earnings to be
Fig. 21.24 | How virtual function calls work.

Flow of Virtual Function Call baseClassPtr->print()
When baseClassPtr Points to Object hourlyEmployee

1. pass &hourlyEmployee to baseClassPtr
2. get to hourlyEmployee object
3. get to HourlyEmployee vtable
4. get to print pointer in vtable
5. execute print for HourlyEmployee
a concrete class, because earnings is a pure virtual function. These classes do not need to implement function print, however, to be considered concrete—print is not a pure virtual function and derived classes can inherit class Employee’s implementation of print. Furthermore, class BasePlusCommissionEmployee does not have to implement either function print or earnings—both function implementations can be inherited from class CommissionEmployee. If a class in our hierarchy were to inherit function implementations in this manner, the vtable pointers for these functions would simply point to the function implementation that was being inherited. For example, if BasePlusCommissionEmployee did not override earnings, the earnings function pointer in the vtable for class BasePlusCommissionEmployee would point to the same earnings function as the vtable for class CommissionEmployee points to.

Polymorphism is accomplished through an elegant data structure involving three levels of pointers. We’ve discussed one level—the function pointers in the vtable. These point to the actual functions that execute when a virtual function is invoked.

Now we consider the second level of pointers. Whenever an object of a class with one or more virtual functions is instantiated, the compiler attaches to the object a pointer to the vtable for that class. This pointer is normally at the front of the object, but it isn’t required to be implemented that way. In Fig. 21.24, these pointers are associated with the objects created in Fig. 21.23 (one object for each of the types SalariedEmployee, HourlyEmployee, CommissionEmployee and BasePlusCommissionEmployee). Notice that the diagram displays each of the object’s data member values. For example, the salariedEmployee object contains a pointer to the SalariedEmployee vtable; the object also contains the values John Smith, 111-11-1111 and $800.00.

The third level of pointers simply contains the handles to the objects that receive the virtual function calls. The handles in this level may also be references. Fig. 21.24 depicts the vector employees that contains Employee pointers.

Now let’s see how a typical virtual function call executes. Consider the call baseClassPtr->print() in function virtualViaPointer (line 75 of Fig. 21.23). Assume that baseClassPtr contains employees[1] (i.e., the address of object hourlyEmployee in employees). When the compiler compiles this statement, it determines that the call is indeed being made via a base-class pointer and that print is a virtual function.

The compiler determines that print is the second entry in each of the vtables. To locate this entry, the compiler notes that it will need to skip the first entry. Thus, the compiler compiles an offset or displacement of four bytes (four bytes for each pointer on today’s popular 32-bit machines, and only one pointer needs to be skipped) into the table of machine-language object-code pointers to find the code that will execute the virtual function call.

The compiler generates code that performs the following operations [Note: The numbers in the list correspond to the circled numbers in Fig. 21.24]:

1. Select the $i^{th}$ entry of employees (in this case, the address of object hourlyEmployee), and pass it as an argument to function virtualViaPointer. This sets parameter baseClassPtr to point to hourlyEmployee.
2. Dereference that pointer to get to the hourlyEmployee object—which, as you recall, begins with a pointer to the HourlyEmployee vtable.
3. Dereference hourlyEmployee’s vtable pointer to get to the HourlyEmployee vtable.
4. Skip the offset of four bytes to select the print function pointer.
5. Dereference the print function pointer to form the “name” of the actual function to execute, and use the function call operator () to execute the appropriate print function, which in this case prints the employee’s type, name, social security number, hourly wage and hours worked.

Fig. 21.24’s data structures may appear to be complex, but this complexity is managed by the compiler and hidden from you, making polymorphic programming straightforward. The pointer dereferencing operations and memory accesses that occur on every virtual function call require some additional execution time. The vtables and the vtable pointers added to the objects require some additional memory. You now have enough information to determine whether virtual functions are appropriate for your programs.

**Performance Tip 21.1**
Polymorphism, as typically implemented with virtual functions and dynamic binding in C++, is efficient. You can use these capabilities with nominal impact on performance.

**Performance Tip 21.2**
Virtual functions and dynamic binding enable polymorphic programming as an alternative to switch logic programming. Optimizing compilers normally generate polymorphic code that runs as efficiently as hand-coded switch-based logic. Polymorphism’s overhead is acceptable for most applications. But in some situations—such as real-time applications with stringent performance requirements—polymorphism’s overhead may be too high.

**Software Engineering Observation 21.11**
Dynamic binding enables independent software vendors (ISVs) to distribute software without revealing proprietary secrets. Software distributions can consist of only header files and object files—no source code needs to be revealed. Software developers can then use inheritance to derive new classes from those provided by the ISVs. Other software that worked with the classes the ISVs provided will still work with the derived classes and will use the overridden virtual functions provided in these classes (via dynamic binding).

### 21.8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, dynamic_cast, typeid and type_info

Recall from the problem statement at the beginning of Section 21.6 that, for the current pay period, our fictitious company has decided to reward BasePlusCommissionEmployees by adding 10 percent to their base salaries. When processing Employee objects polymorphically in Section 21.6.6, we did not need to worry about the “specifics.” Now, however, to adjust the base salaries of BasePlusCommissionEmployees, we have to determine the specific type of each Employee object at execution time, then act appropriately. This section demonstrates the powerful capabilities of runtime type information (RTTI) and dynamic casting, which enable a program to determine the type of an object at execution time and act on that object accordingly.

[Note: Some compilers require that RTTI be enabled before it can be used in a program. In Visual C++ 2008, this option is enabled by default.]
Figure 21.25 uses the Employee hierarchy developed in Section 21.6 and increases by 10 percent the base salary of each BasePlusCommissionEmployee. Line 22 declares four-element vector employees that stores pointers to Employee objects. Lines 25–32 populate the vector with the addresses of dynamically allocated objects of classes SalariedEmployee (Figs. 21.15–21.16), HourlyEmployee (Figs. 21.17–21.18), CommissionEmployee (Figs. 21.19–21.20) and BasePlusCommissionEmployee (Figs. 21.21–21.22).

```cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include "Employee.h"
#include "SalariedEmployee.h"
#include "HourlyEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main()
{
    // set floating-point output formatting
    cout << fixed << setprecision(2);

    // create vector of four base-class pointers
    vector < Employee * > employees(4);

    // initialize vector with various kinds of Employees
    employees[0] = new SalariedEmployee("John", "Smith", "111-11-1111", 800);
    employees[1] = new HourlyEmployee("Karen", "Price", "222-22-2222", 16.75, 40);
    employees[2] = new CommissionEmployee("Sue", "Jones", "333-33-3333", 10000, .06);
    employees[3] = new BasePlusCommissionEmployee("Bob", "Lewis", "444-44-4444", 5000, .04, 300);

    // polymorphically process each element in vector employees
    for (size_t i = 0; i < employees.size(); i++)
    {
        employees[i]->print(); // output employee information
        cout << endl;
    }

    // downcast pointer
    BasePlusCommissionEmployee *derivedPtr =
        dynamic_cast < BasePlusCommissionEmployee * > (employees[i]);
}
```

Fig. 21.25 | Demonstrating downcasting and runtime type information. (Part 1 of 2.)
The for statement in lines 35–57 iterates through the employees vector and displays each Employee’s information by invoking member function print (line 37). Recall that
because print is declared virtual in base class Employee, the system invokes the appropriate derived-class object's print function.

In this example, as we encounter BasePlusCommissionEmployee objects, we wish to increase their base salary by 10 percent. Since we process the employees generically (i.e., polymorphically), we cannot (with the techniques we've learned) be certain as to which type of Employee is being manipulated at any given time. This creates a problem, because BasePlusCommissionEmployee employees must be identified when we encounter them so they can receive the 10 percent salary increase. To accomplish this, we use operator dynamic_cast (line 42) to determine whether the type of each object is BasePlusCommissionEmployee. This is the downcast operation we referred to in Section 21.3.3. Lines 41–43 dynamically downcast employees[i] from type Employee * to type BasePlusCommissionEmployee *. If the vector element points to an object that is a BasePlusCommissionEmployee object, then that object's address is assigned to commissionPtr; otherwise, 0 is assigned to derived-class pointer derivedPtr.

If the value returned by the dynamic_cast operator in lines 41–43 is not 0, the object is the correct type, and the if statement (lines 47–54) performs the special processing required for the BasePlusCommissionEmployee object. Lines 49, 51 and 53 invoke BasePlusCommissionEmployee functions getBaseSalary and setBaseSalary to retrieve and update the employee's salary.

Line 56 invokes member function earnings on the object to which employees[i] points. Recall that earnings is declared virtual in the base class, so the program invokes the derived-class object's earnings function—another example of dynamic binding.

Lines 60–67 display each employee's object type and uses the delete operator to deallocate the dynamic memory to which each vector element points. Operator typeid (line 64) returns a reference to an object of class type_info that contains the information about the type of its operand, including the name of that type. When invoked, type_info member function name (line 64) returns a pointer-based string that contains the type name (e.g., "class BasePlusCommissionEmployee") of the argument passed to typeid. To use typeid, the program must include header file <typeinfo> (line 8).

Portability Tip 21.1

The string returned by type_info member function name may vary by compiler.

We avoid several compilation errors in this example by downcasting an Employee pointer to a BasePlusCommissionEmployee pointer (lines 41–43). If we remove the dynamic_cast from line 42 and attempt to assign the current Employee pointer directly to BasePlusCommissionEmployee pointer derivedPtr, we'll receive a compilation error. C++ does not allow a program to assign a base-class pointer to a derived-class pointer because the is-a relationship does not apply—a CommissionEmployee is not a BasePlusCommissionEmployee. The is-a relationship applies only between the derived class and its base classes, not vice versa.

Similarly, if lines 49, 51 and 53 used the current base-class pointer from employees, rather than derived-class pointer derivedPtr, to invoke derived-class-only functions getBaseSalary and setBaseSalary, we'd receive a compilation error at each of these lines. As you learned in Section 21.3.3, attempting to invoke derived-class-only functions through a base-class pointer is not allowed. Although lines 49, 51 and 53 execute only if
commissionPtr is not 0 (i.e., if the cast can be performed), we cannot attempt to invoke derived-class BasePlusCommissionEmployee functions getBaseSalary and setBaseSalary on the base-class Employee pointer. Recall that, using a base class Employee pointer, we can invoke only functions found in base class Employee—earnings, print and Employee’s get and set functions.

21.9 Virtual Destructors

A problem can occur when using polymorphism to process dynamically allocated objects of a class hierarchy. So far you’ve seen nonvirtual destructors—destructors that are not declared with keyword virtual. If a derived-class object with a nonvirtual destructor is destroyed explicitly by applying the delete operator to a base-class pointer to the object, the C++ standard specifies that the behavior is undefined.

The simple solution to this problem is to create a virtual destructor (i.e., a destructor that is declared with keyword virtual) in the base class. This makes all derived-class destructors virtual even though they do not have the same name as the base-class destructor. Now, if an object in the hierarchy is destroyed explicitly by applying the delete operator to a base-class pointer, the destructor for the appropriate class is called based on the object to which the base-class pointer points. Remember, when a derived-class object is destroyed, the base-class part of the derived-class object is also destroyed, so it’s important for the destructors of both the derived class and base class to execute. The base-class destructor automatically executes after the derived-class destructor.

Error-Prevention Tip 21.2

If a class has virtual functions, provide a virtual destructor, even if one is not required for the class. This ensures that a custom derived-class destructor (if there is one) will be invoked when a derived-class object is deleted via a base class pointer.

Common Programming Error 21.5

Constructors cannot be virtual. Declaring a constructor virtual is a compilation error.

21.10 Wrap-Up

In this chapter we discussed polymorphism, which enables us to “program in the general” rather than “program in the specific,” and we showed how this makes programs more extensible. We began with an example of how polymorphism would allow a screen manager to display several “space” objects. We then demonstrated how base-class and derived-class pointers can be aimed at base-class and derived-class objects. We said that aiming base-class pointers at base-class objects is natural, as is aiming derived-class pointers at derived-class objects. Aiming base-class pointers at derived-class objects is also natural because a derived-class object is an object of its base class. You learned why aiming derived-class pointers at base-class objects is dangerous and why the compiler disallows such assignments. We introduced virtual functions, which enable the proper functions to be called when objects at various levels of an inheritance hierarchy are referenced (at execution time) via base-class pointers. This is known as dynamic or late binding. We then discussed pure virtual functions (virtual functions that do not provide an implementation) and abstract classes
(classes with one or more pure virtual functions). You learned that abstract classes cannot
be used to instantiate objects, while concrete classes can. We then demonstrated using ab-
stract classes in an inheritance hierarchy. You learned how polymorphism works “under the
hood” with vtables that are created by the compiler. We used runtime type information
(RTTI) and dynamic casting to determine the type of an object at execution time and act
on that object accordingly. The chapter concluded with a discussion of virtual destruct-
tors, and how they ensure that all appropriate destructors in an inheritance hierarchy run
on a derived-class object when that object is deleted via a base-class pointer.
In the next chapter, we discuss templates, a sophisticated feature of C++ that enables
you to define a family of related classes or functions with a single code segment.

Summary

Section 21.1 Introduction
• Polymorphism enables us to “program in the general” rather than “program in the specific.”
• Polymorphism enables us to write programs that process objects of classes that are part of the
same class hierarchy as if they were all objects of the hierarchy’s base class.
• With polymorphism, we can design and implement systems that are easily extensible—new class-
es can be added with little or no modification to the general portions of the program. The only
parts of a program that must be altered to accommodate new classes are those that require direct
knowledge of the new classes that you add to the hierarchy.
• Runtime type information (RTTI) and dynamic casting enable a program to determine the type
of an object at execution time and act on that object accordingly.

Section 21.2 Polymorphism Examples
• With polymorphism, one function can cause different actions to occur, depending on the type
of the object on which the function is invoked.
• This makes it possible to design and implement more extensible systems. Programs can be writ-
ten to process objects of types that may not exist when the program is under development.

Section 21.3 Relationships Among Objects in an Inheritance Hierarchy
• C++ enables polymorphism—the ability for objects of different classes related by inheritance to
respond differently to the same member-function call.
• Polymorphism is implemented via virtual functions and dynamic binding.
• When a base-class pointer or reference is used to call a virtual function, C++ chooses the correct
overridden function in the appropriate derived class associated with the object.
• If a virtual function is called by referencing a specific object by name and using the dot mem-
ber-selection operator, the reference is resolved at compile time (this is called static binding); the
virtual function that is called is the one defined for the class of that particular object.
• Derived classes can provide their own implementations of a base-class virtual function if nec-
essary, but if they do not, the base class’s implementation is used.

Section 21.4 Type Fields and switch Statements
• Polymorphic programming with virtual functions can eliminate the need for switch logic. You
can use the virtual function mechanism to perform the equivalent logic automatically, thus
avoiding the kinds of errors typically associated with switch logic.
Section 21.5 Abstract Classes and Pure virtual Functions
• Abstract classes are typically used as base classes, so we refer to them as abstract base classes. No objects of an abstract class may be instantiated.
• Classes from which objects can be instantiated are concrete classes.
• You create an abstract class by declaring one or more pure virtual functions with pure specifiers (= 0) in their declarations.
• If a class is derived from a class with a pure virtual function and that derived class does not supply a definition for that pure virtual function, then that virtual function remains pure in the derived class. Consequently, the derived class is also an abstract class.
• Although we cannot instantiate objects of abstract base classes, we can declare pointers and references to objects of abstract base classes. Such pointers and references can be used to enable polymorphic manipulations of derived-class objects instantiated from concrete derived classes.

Section 21.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”
• Dynamic binding requires that at runtime, the call to a virtual member function be routed to the virtual function version appropriate for the class. A virtual function table called the vtable is implemented as an array containing function pointers. Each class with virtual functions has a vtable. For each virtual function in the class, the vtable has an entry containing a function pointer to the version of the virtual function to use for an object of that class. The virtual function to use for a particular class could be the function defined in that class, or it could be a function inherited either directly or indirectly from a base class higher in the hierarchy.
• When a base class provides a virtual member function, derived classes can override the virtual function, but they do not have to override it.
• Each object of a class with virtual functions contains a pointer to the vtable for that class. When a function call is made from a base-class pointer to a derived-class object, the appropriate function pointer in the vtable is obtained and dereferenced to complete the call at execution time.
• Any class that has one or more 0 pointers in its vtable is an abstract class. Classes without any vtable pointers are concrete classes.
• New kinds of classes are regularly added to systems and accommodated by dynamic binding.

Section 21.8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, dynamic_cast, typeid and type_info
• Operator dynamic_cast checks the type of the object to which a pointer points, then determines whether the type has an is-a relationship with the type to which the pointer is being converted. If so, dynamic_cast returns the object’s address. If not, dynamic_cast returns 0.
• Operator typeid returns a reference to a type_info object that contains information about the operand’s type, including the type name. To use typeid, the program must include header file <typeinfo>.
• When invoked, type_info member function name returns a pointer-based string that contains the name of the type that the type_info object represents.
• Operators dynamic_cast and typeid are part of C++’s runtime type information (RTTI) feature, which allows a program to determine an object’s type at runtime.

Section 21.9 Virtual Destructors
• Declare the base-class destructor virtual if the class contains virtual functions. This makes all derived-class destructors virtual, even though they do not have the same name as the base-class destructor. If an object in the hierarchy is destroyed explicitly by applying the delete operator
to a base-class pointer to a derived-class object, the destructor for the appropriate class is called. After a derived-class destructor runs, the destructors for all of that class’s base classes run all the way up the hierarchy.

**Terminology**

abstract base classes 799  
abstract classes 799  
concrete classes 799  
displacement into a vtable 821  
downcasting 792  
dynamic binding 793  
dynamic casting 780  
dynamic_cast 825  
implementation inheritance 802  
interface inheritance 802  
iterator class 801  
late binding 793  
name function of class type_info 825  
nonvirtual destructor 826  
offset into a vtable 821  
override 792  
polymorphism 779  
pure specifier (with virtual functions) 800  
pure virtual function 800  
runtime type information (RTTI) 780  
static binding 793  
typeid operator 825  
type_info class 825  
<typename> header file 825  
virtual destructor 826  
virtual function 792  
virtual function table (vtable) 819

**Self-Review Exercises**

21.1 Fill in the blanks in each of the following statements:

a) Treating a base-class object as a(n) _______ can cause errors.

b) Polymorphism helps eliminate _______ logic.

c) If a class contains at least one pure virtual function, it’s a(n) _______ class.

d) Classes from which objects can be instantiated are called _______ classes.

e) Operator _______ can be used to downcast base-class pointers safely.

f) Operator typeid returns a reference to a(n) _______ object.

g) _______ involves using a base-class pointer or reference to invoke virtual functions on base-class and derived-class objects.

h) Overridable functions are declared using keyword _______.

i) Casting a base-class pointer to a derived-class pointer is called _______.

21.2 State whether each of the following is true or false. If false, explain why.

a) All virtual functions in an abstract base class must be declared as pure virtual functions.

b) Referring to a derived-class object with a base-class handle is dangerous.

c) A class is made abstract by declaring that class virtual.

d) If a base class declares a pure virtual function, a derived class must implement that function to become a concrete class.

e) Polymorphic programming can eliminate the need for switch logic.

**Answers to Self-Review Exercises**

21.1  a) derived-class object. b) switch. c) abstract. d) concrete. e) dynamic_cast. f) type_info.

g) Polymorphism. h) virtual. i) downcasting.

21.2  a) False. An abstract base class can include virtual functions with implementations.  
b) False. Referring to a base-class object with a derived-class handle is dangerous.  
c) False. Classes are never declared virtual. Rather, a class is made abstract by including at least one pure virtual function in the class.  
d) True.  
e) True.
Chapter 21 Object-Oriented Programming: Polymorphism

Exercises

21.3 How is it that polymorphism enables you to program “in the general” rather than “in the specific”? Discuss the key advantages of programming “in the general.”

21.4 Discuss the problems of programming with switch logic. Explain why polymorphism can be an effective alternative to using switch logic.

21.5 Distinguish between inheriting interface and inheriting implementation. How do inheritance hierarchies designed for inheriting interface differ from those designed for inheriting implementation?

21.6 What are virtual functions? Describe a circumstance in which virtual functions would be appropriate.

21.7 Distinguish between static binding and dynamic binding. Explain the use of virtual functions and the vtable in dynamic binding.

21.8 Distinguish between virtual functions and pure virtual functions.

21.9 (Abstract Base Classes) Suggest one or more levels of abstract base classes for the Shape hierarchy discussed in this chapter and shown in Fig. 20.3. (The first level is Shape, and the second level consists of the classes TwoDimensionalShape and ThreeDimensionalShape.)

21.10 How does polymorphism promote extensibility?

21.11 You’ve been asked to develop a flight simulator that will have elaborate graphical outputs. Explain why polymorphic programming could be especially effective for a problem of this nature.

21.12 (Payroll System Modification) Modify the payroll system of Figs. 21.13–21.23 to include private data member birthDate in class Employee. Use class date from Figs. 19.9–19.10 to represent an employee’s birthday. Assume that payroll is processed once per month. Create a vector of Employee references to store the various employee objects. In a loop, calculate the payroll for each Employee (polymorphically), and add a $100.00 bonus to the person’s payroll amount if the current month is the month in which the Employee’s birthday occurs.

21.13 (Shape Hierarchy) Implement the Shape hierarchy designed in Exercise 20.7 (which is based on the hierarchy in Fig. 20.3). Each TwoDimensionalShape should contain function getArea to calculate the area of the two-dimensional shape. Each ThreeDimensionalShape should have member functions getArea and getVolume to calculate the surface area and volume, respectively, of the three-dimensional shape. Create a program that uses a vector of Shape pointers to objects of each concrete class in the hierarchy. The program should print the object to which each vector element points. Also, in the loop that processes all the shapes in the vector, determine whether each shape is a TwoDimensionalShape or a ThreeDimensionalShape. If a shape is a TwoDimensionalShape, display its area. If a shape is a ThreeDimensionalShape, display its area and volume.

21.14 (Project: Polymorphic Screen Manager Using Shape Hierarchy) Develop a basic graphics package. Use the Shape hierarchy implemented in Exercise 21.13. Limit yourself to two-dimensional shapes such as squares, rectangles, triangles and circles. Interact with the user. Let the user specify the position, size, shape and fill characters to be used in drawing each shape. The user can specify more than one of the same shape. As you create each shape, place a Shape * pointer to each new Shape object into an array. Each Shape class should now have its own draw member function. Write a polymorphic screen manager that walks through the array, sending draw messages to each object in the array to form a screen image. Redraw the screen image each time the user specifies an additional shape.

21.15 (Package Inheritance Hierarchy) Use the Package inheritance hierarchy created in Exercise 20.9 to create a program that displays the address information and calculates the shipping costs for several Packages. The program should contain a vector of Package pointers to objects of
classes TwoDayPackage and OvernightPackage. Loop through the vector to process the Packages polymorphically. For each Package, invoke get functions to obtain the address information of the sender and the recipient, then print the two addresses as they would appear on mailing labels. Also, call each Package’s calculateCost member function and print the result. Keep track of the total shipping cost for all Packages in the vector, and display this total when the loop terminates.

21.16 (Polymorphic Banking Program Using Account Hierarchy) Develop a polymorphic banking program using the Account hierarchy created in Exercise 20.10. Create a vector of Account pointers to SavingsAccount and CheckingAccount objects. For each Account in the vector, allow the user to specify an amount of money to withdraw from the Account using member function debit and an amount of money to deposit into the Account using member function credit. As you process each Account, determine its type. If an Account is a SavingsAccount, calculate the amount of interest owed to the Account using member function calculateInterest, then add the interest to the account balance using member function credit. After processing an Account, print the updated account balance obtained by invoking base-class member function getBalance.

Making a Difference

21.17 (CarbonFootprint Abstract Class: Polymorphism) Using an abstract class with only pure virtual functions, you can specify similar behaviors for possibly disparate classes. Governments and companies worldwide are becoming increasingly concerned with carbon footprints (annual releases of carbon dioxide into the atmosphere) from buildings burning various types of fuels for heat, vehicles burning fuels for power, and the like. Many scientists blame these greenhouse gases for the phenomenon called global warming. Create three small classes unrelated by inheritance—classes Building, Car and Bicycle. Give each class some unique appropriate attributes and behaviors that it does not have in common with other classes. Write an abstract class CarbonFootprint with only a pure virtual getCarbonFootprint method. Have each of your classes inherit from that abstract class and implement the getCarbonFootprint method to calculate an appropriate carbon footprint for that class (check out a few websites that explain how to calculate carbon footprints). Write an application that creates objects of each of the three classes, places pointers to those objects in a vector of CarbonFootprint pointers, then iterates through the vector, polymorphically invoking each object’s getCarbonFootprint method. For each object, print some identifying information and the object’s carbon footprint.
Behind that outside pattern the dim shapes get clearer every day.
It is always the same shape, only very numerous.
—Charlotte Perkins Gilman

Every man of genius sees the world at a different angle from his fellows.
—Havelock Ellis

...our special individuality, as distinguished from our generic humanity.
—Oliver Wendell Holmes, Sr.

**Objectives**

In this chapter you’ll learn:

- To use function templates to conveniently create a group of related (overloaded) functions.
- To distinguish between function templates and function-template specializations.
- To use class templates to create groups of related types.
- To distinguish between class templates and class-template specializations.
- To overload function templates.
- To understand the relationships among templates, friends, inheritance and static members.
22.1 Introduction

In this chapter, we discuss one of C++’s more powerful software reuse features, namely templates. Function templates and class templates enable you to specify, with a single code segment, an entire range of related (overloaded) functions—called function-template specializations—or an entire range of related classes—called class-template specializations. This technique is called generic programming.

We might write a single function template for an array-sort function, then have C++ generate separate function-template specializations that will sort *int* arrays, *float* arrays, *string* arrays and so on. We introduced function templates in Chapter 15. We present an additional discussion and example in this chapter.

We might write a single class template for a stack class, then have C++ generate separate class-template specializations, such as a stack-of-*int* class, a stack-of-*float* class, a stack-of-*string* class and so on.

Note the distinction between templates and template specializations: Function templates and class templates are like stencils out of which we trace shapes; function-template specializations and class-template specializations are like the separate tracings that all have the same shape, but could, for example, be drawn in different colors.

In this chapter, we present a function template and a class template. We also consider the relationships between templates and other C++ features, such as overloading, inheritance, friends and *static* members. The design and details of the template mechanisms discussed here are based on the work of Bjarne Stroustrup as presented in his paper, “Parameterized Types for C++”—published in the *Proceedings of the USENIX C++ Conference* held in Denver, Colorado, in October 1988.

**Software Engineering Observation 22.1**

Most C++ compilers require the complete definition of a template to appear in the client source-code file that uses the template. For this reason and for reusability, templates are often defined in header files, which are then #included into the appropriate client source-code files. For class templates, this means that the member functions are also defined in the header file.

22.2 Function Templates

Overloaded functions normally perform similar or identical operations on different types of data. If the operations are identical for each type, they can be expressed more compactly
and conveniently using function templates. Initially, you write a single function-template definition. Based on the argument types provided explicitly or inferred from calls to this function, the compiler generates separate source-code functions (i.e., function-template specializations) to handle each function call appropriately. In C, this task can be performed using macros created with the preprocessor directive \texttt{#define} (see Chapter 13). However, macros can have serious side effects and do not enable the compiler to perform type checking. Function templates provide a compact solution, like macros, but enable full type checking.

### Error-Prevention Tip 22.1
Function templates, like macros, enable software reuse. Unlike macros, function templates help eliminate many types of errors through the scrutiny of full C++ type checking.

All function-template definitions begin with keyword \texttt{template} followed by a list of template parameters to the function template enclosed in angle brackets (< and >); each template parameter that represents a type must be preceded by either of the interchangeable keywords \texttt{class} or \texttt{typename}, as in

```cpp
template< typename T >
```

or

```cpp
template< class ElementType >
```

or

```cpp
template< typename BorderType, typename FillType >
```

The type template parameters of a function-template definition are used to specify the types of the arguments to the function, to specify the return type of the function and to declare variables within the function. The function definition follows and appears like any other function definition. Keywords \texttt{typename} and \texttt{class} used to specify function-template parameters actually mean “any fundamental type or user-defined type.”

### Common Programming Error 22.1
Not placing keyword \texttt{class} or keyword \texttt{typename} before each type template parameter of a function template is a syntax error.

### Example: Function Template \texttt{printArray}
Let’s examine function template \texttt{printArray} in Fig. 22.1, lines 7–14. Function template \texttt{printArray} declares (line 7) a single template parameter \texttt{T} (\texttt{T} can be any valid identifier) for the type of the array to be printed by function \texttt{printArray}; \texttt{T} is referred to as a type template parameter, or type parameter. You’ll see nontype template parameters in Section 22.5.

```cpp
// Fig. 22.1: fig22_01.cpp
// Using template functions.
#include <iostream>
```

Fig. 22.1 | Function-template specializations of function template \texttt{printArray}. (Part 1 of 2.)
When the compiler detects a `printArray` function invocation in the client program (e.g., lines 29, 34 and 39), the compiler uses its overload resolution capabilities to find a definition of function `printArray` that best matches the function call. In this case, the only `printArray` function with the appropriate number of parameters is the `printArray` function template (lines 7–14). Consider the function call at line 29. The compiler com-
Chapter 22  Templates

parses the type of printArray's first argument (int * at line 29) to the printArray function template's first parameter (const T * const at line 8) and deduces that replacing the type parameter T with int would make the argument consistent with the parameter. Then, the compiler substitutes int for T throughout the template definition and compiles a printArray specialization that can display an array of int values. In Fig. 22.1, the compiler creates three printArray specializations—one that expects an int array, one that expects a double array and one that expects a char array. For example, the function-template specialization for type int is

```cpp
void printArray( const int * const array, int count )
{
    for ( int i = 0; i < count; i++ )
        cout << array[ i ] << " ";
    cout << endl;
} // end function printArray
```

As with function parameters, the names of template parameters must be unique inside a template definition. Template parameter names need not be unique across different function templates.

Figure 22.1 demonstrates function template printArray (lines 7–14). The program begins by declaring five-element int array a, seven-element double array b and six-element char array c (lines 22–24, respectively). Then, the program outputs each array by calling printArray—once with a first argument a of type int * (line 29), once with a first argument b of type double * (line 34) and once with a first argument c of type char * (line 39). The call in line 29, for example, causes the compiler to infer that T is int and to instantiate a printArray function-template specialization, for which type parameter T is int. The call in line 34 causes the compiler to infer that T is double and to instantiate a second printArray function-template specialization, for which type parameter T is double. The call in line 39 causes the compiler to infer that T is char and to instantiate a third printArray function-template specialization, for which type parameter T is char. It’s important to note that if T (line 7) represents a user-defined type (which it does not in Fig. 22.1), there must be an overloaded stream insertion operator for that type; otherwise, the first stream insertion operator in line 11 will not compile.

**Common Programming Error 22.2**

If a template is invoked with a user-defined type, and if that template uses functions or operators (e.g., ==, +, <=) with objects of that class type, then those functions and operators must be overloaded for the user-defined type. Forgetting to overload such operators causes compilation errors.

In this example, the template mechanism saves you from having to write three separate overloaded functions with prototypes

```cpp
void printArray( const int * const, int );
void printArray( const double * const, int );
void printArray( const char * const, int );
```

that all use the same code, except for type T (as used in line 8).
22.3 Overloading Function Templates

Function templates and overloading are intimately related. The function-template specializations generated from a function template all have the same name, so the compiler uses overloading resolution to invoke the proper function.

A function template may be overloaded in several ways. We can provide other function templates that specify the same function name but different function parameters. For example, function template `printArray` of Fig. 22.1 could be overloaded with another `printArray` function template with additional parameters `lowSubscript` and `highSubscript` to specify the portion of the array to output (see Exercise 22.4).

A function template also can be overloaded by providing nontemplate functions with the same function name but different function arguments. For example, function template `printArray` of Fig. 22.1 could be overloaded with a nontemplate version that specifically prints an array of character strings in neat, tabular format (see Exercise 22.5).

The compiler performs a matching process to determine what function to call when a function is invoked. First, the compiler tries to find and use a precise match in which the function names and argument types are consistent with those of the function call. If this fails, the compiler determines whether a function template is available that can be used to generate a function-template specialization with a precise match of function name and argument types. If such a function template is found, the compiler generates and uses the appropriate function-template specialization. If not, the compiler generates an error message. Also, if there are multiple matches for the function call, the compiler considers the call to be ambiguous and the compiler generates an error message.

22.4 Class Templates

It’s possible to understand the concept of a “stack” (a data structure into which we insert items at the top and retrieve those items in last-in, first-out order) independent of the type of the items being placed in the stack. However, to instantiate a stack, a data type must be specified. This creates a wonderful opportunity for software reusability. We need the means for describing the notion of a stack generically and instantiating classes that are type-specific versions of this generic stack class. C++ provides this capability through class templates.
Class templates are called parameterized types, because they require one or more type parameters to specify how to customize a “generic class” template to form a class-template specialization.

To produce a variety of class-template specializations you write only one class-template definition. Each time an additional class-template specialization is needed, you use a concise, simple notation, and the compiler writes the source code for the specialization you require. One Stack class template, for example, could thus become the basis for creating many Stack classes (such as “Stack of double,” “Stack of int,” “Stack of char,” “Stack of Employee,” etc.) used in a program.

**Creating Class Template Stack< T >**

Note the Stack class-template definition in Fig. 22.2. It looks like a conventional class definition, except that it’s preceded by the header (line 6)

```cpp
template< typename T >
```

to specify a class-template definition with type parameter T which acts as a placeholder for the type of the Stack class to be created. You need not specifically use identifier T—any valid identifier can be used. The type of element to be stored on this Stack is mentioned generically as T throughout the Stack class header and member-function definitions. In a moment, we show how T becomes associated with a specific type, such as double or int. Due to the way this class template is designed, there are two constraints for nonfundamental data types used with this Stack—they must have a default constructor (for use in line 44 to create the array that stores the stack elements), and their assignment operators must properly copy objects into the Stack (lines 56 and 70).

```cpp
// Fig. 22.2: Stack.h
// Stack class template.
#ifndef STACK_H
#define STACK_H

template< typename T >
class Stack
{
    public:
        Stack( int = 10 ); // default constructor (Stack size 10)
    ~Stack() { delete [] stackPtr; // deallocate internal space for Stack }
    bool push( const T & ); // push an element onto the Stack
    bool pop( T & ); // pop an element off the Stack
}
#endif
```

**Software Engineering Observation 22.2**

Class templates encourage software reusability by enabling type-specific versions of generic classes to be instantiated.
22.4 Class Templates

// determine whether Stack is empty
bool isEmpty() const
{
    return top == -1;
} // end function isEmpty

// determine whether Stack is full
bool isFull() const
{
    return top == size - 1;
} // end function isFull

private:

    int size; // # of elements in the Stack
    int top; // location of the top element (-1 means empty)
    T *stackPtr; // pointer to internal representation of the Stack
}; // end class template Stack

// constructor template

template<typename T>
Stack< T >::Stack( int s )
    : size( s > 0 ? s : 10 ), // validate size
top( -1 ), // Stack initially empty
stackPtr( new T[size] ) // allocate memory for elements
{
    // empty body
} // end Stack constructor template

// push element onto Stack;
// if successful, return true; otherwise, return false

template<typename T>
bool Stack< T >::push( const T &pushValue )
{
    if ( !isFull() )
    {
        stackPtr[ ++top ] = pushValue; // place item on Stack
        return true; // push successful
    } // end if

    return false; // push unsuccessful
} // end function template push

// pop element off Stack;
// if successful, return true; otherwise, return false

template<typename T>
bool Stack< T >::pop( T &popValue )
{
    if ( !isEmpty() )
    {
        popValue = stackPtr[ top-- ]; // remove item from Stack
        return true; // pop successful
    } // end if

Fig. 22.2  Class template Stack. (Part 2 of 3.)
Chapter 22 Templates

The member-function definitions of a class template are function templates. The member-function definitions that appear outside the class template definition each begin with the header (lines 40, 51 and 65). Thus, each definition resembles a conventional function definition, except that the Stack element type always is listed generically as type parameter $T$. The binary scope resolution operator is used with the class-template name $Stack< T >$ (lines 41, 52 and 66) to tie each member-function definition to the class template’s scope. In this case, the generic class name is $Stack< T >$. When $doubleStack$ is instantiated as type $Stack<double>$, the Stack constructor function-template specialization uses new to create an array of elements of type $double$ to represent the stack (line 44). The statement

```
stackPtr( new $T[ size ] );
```

in the Stack class-template definition is generated by the compiler in the class-template specialization $Stack<double>$ as

```
stackPtr( new $double[ size ] );
```

**Creating a Driver to Test Class Template $Stack< T >$**

Now, let’s consider the driver (Fig. 22.3) that exercises the $Stack$ class template. The driver begins by instantiating object $doubleStack$ of size 5 (line 9). This object is declared to be of class $Stack<double>$ (pronounced "Stack of double"). The compiler associates type $double$ with type parameter $T$ in the class template to produce the source code for a Stack class of type $double$. Although templates offer software-reusability benefits, remember that multiple class-template specializations are instantiated in a program (at compile time), even though the template is written only once.

```
// Fig. 22.3: fig22_03.cpp
// Stack class template test program.
#include <iostream>
#include "Stack.h" // Stack class template definition
using namespace std;

int main()
{
    $Stack< double >$ doubleStack( 5 ); // size 5
    double doubleValue = 1.1;
}```
cout << "Pushing elements onto doubleStack\n";

// push 5 doubles onto doubleStack
while ( doubleStack.push( doubleValue ) )
{
    cout << doubleValue << ' ';
    doubleValue += 1.1;
} // end while

cout << "Stack is full. Cannot push " << doubleValue
     << "\n\n\nPopping elements from doubleStack\n";

// pop elements from doubleStack
while ( doubleStack.pop( doubleValue ) )
    cout << doubleValue << ' ';

cout << "Stack is empty. Cannot pop\n";

Stack<int> intStack; // default size 10
int intValue = 1;
cout << "Pushing elements onto intStack\n";

// push 10 integers onto intStack
while ( intStack.push( intValue ) )
{
    cout << intValue++ << ' ';
} // end while

cout << "Stack is full. Cannot push " << intValue
     << "\n\n\nPopping elements from intStack\n";

// pop elements from intStack
while ( intStack.pop( intValue ) )
cout << intValue << ' ';

cout << "Stack is empty. Cannot pop" << endl;
} // end main

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop

Fig. 22.3 | Class template Stack test program. (Part 2 of 2.)
Lines 15–19 invoke push to place the double values 1.1, 2.2, 3.3, 4.4 and 5.5 onto doubleStack. The while loop terminates when the driver attempts to push a sixth value onto doubleStack (which is full, because it holds a maximum of five elements). Function push returns false when it’s unable to push a value onto the stack.\(^1\)

Lines 25–26 invoke pop in a while loop to remove the five values from the stack (note, in the output of Fig. 22.3, that the values do pop off in last-in, first-out order). When the driver attempts to pop a sixth value, the doubleStack is empty, so the pop loop terminates.

Line 30 instantiates integer stack intStack with the declaration

```
Stack< int > intStack;
```

(pronounced “intStack is a Stack of int”). Because no size is specified, the size defaults to 10 as specified in the default constructor (Fig. 22.2, line 10). Lines 35–38 loop and invoke push to place values onto intStack until it’s full, then lines 44–45 loop and invoke pop to remove values from intStack until it’s empty. Once again, notice in the output that the values pop off in last-in, first-out order.

Creating Function Templates to Test Class Template \texttt{Stack< T >}

Notice that the code in function \texttt{main} of Fig. 22.3 is almost identical for both the double-Stack manipulations in lines 9–28 and the intStack manipulations in lines 30–47. This presents another opportunity to use a function template. Figure 22.4 defines function template \texttt{testStack} (lines 10–34) to perform the same tasks as \texttt{main} in Fig. 22.3—push a series of values onto a \texttt{Stack< T >} and pop the values off a \texttt{Stack< T >}. Function template \texttt{testStack} uses template parameter \texttt{T} (specified at line 10) to represent the data type stored in the \texttt{Stack< T >}. The function template takes four arguments (lines 12–15)—a reference to an object of type \texttt{Stack< T >}, a value of type \texttt{T} that will be the first value pushed onto the \texttt{Stack< T >}, a value of type \texttt{T} used to increment the values pushed onto the \texttt{Stack< T >} and a string that represents the name of the \texttt{Stack< T >} object for output purposes. Function \texttt{main} (lines 36–43) instantiates an object of type \texttt{Stack< double >} called doubleStack (line 38) and an object of type \texttt{Stack< int >} called intStack (line 39) and uses these objects in lines 41 and 42. The compiler infers the type of \texttt{T} for \texttt{testStack} from the type used to instantiate the function’s first argument (i.e., the type used to instantiate doubleStack or intStack). The output of Fig. 22.4 precisely matches the output of Fig. 22.3.

1. Class \texttt{Stack} (Fig. 22.2) provides the function \texttt{isFull}, which you can use to determine whether the stack is full before attempting a push operation. This would avoid the potential error of pushing onto a full stack. As we discuss in Chapter 24, Exception Handling, if the operation cannot be completed, function \texttt{push} would “throw an exception.” You can write code to “catch” that exception, then decide how to handle it appropriately for the application. The same technique can be used with function \texttt{pop} when an attempt is made to pop an element from an empty stack.
```cpp
using namespace std;

// function template to manipulate Stack< T >
template< typename T >
void testStack(
    Stack< T > &theStack, // reference to Stack< T >
    T value, // initial value to push
    T increment, // increment for subsequent values
    const string stackName ) // name of the Stack< T > object
{
    cout << "\nPushing elements onto " << stackName << '\n';

    // push element onto Stack
    while ( theStack.push( value ) )
    {
        cout << value << ' ';
        value += increment;
    } // end while

    cout << "\nStack is full. Cannot push " << value
    << "\n\nPopping elements from " << stackName << '\n';

    // pop elements from Stack
    while ( theStack.pop( value ) )
    {
        cout << value << ' ';
    }
    cout << "\nStack is empty. Cannot pop" << endl;
}

int main()
{
    Stack< double > doubleStack( 5 ); // size 5
    Stack< int > intStack; // default size 10
    testStack( doubleStack, 1.1, 1.1, "doubleStack" );
    testStack( intStack, 1, 1, "intStack" );
} // end main
```

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop

**Fig. 22.4** | Passing a Stack template object to a function template. (Part 2 of 2.)
22.5 Nontype Parameters and Default Types for Class Templates

Class template Stack of Section 22.4 used only a type parameter in the template header (Fig. 22.2, line 6). It’s also possible to use nontype template parameters, which can have default arguments and are treated as consts. For example, the template header could be modified to take an int elements parameter as follows:

```cpp
template< typename T, int elements > // nontype parameter elements
```

Then, a declaration such as

```cpp
Stack< double, 100 > mostRecentSalesFigures;
```

could be used to instantiate (at compile time) a 100-element Stack class-template specialization of double values named mostRecentSalesFigures; this class-template specialization would be of type Stack< double, 100 >. The class definition then might contain a private data member with an array declaration such as

```cpp
T stackHolder[ elements ]; // array to hold Stack contents
```

In addition, a type parameter can specify a default type. For example,

```cpp
template< typename T = string > // defaults to type string
```

might specify that a Stack contains string objects by default. Then, a declaration such as

```cpp
Stack<> jobDescriptions;
```

could be used to instantiate a Stack class-template specialization of strings named jobDescriptions; this class-template specialization would be of type Stack< string >. Default type parameters must be the rightmost (trailing) parameters in a template’s type-parameter list. When one is instantiating a class with two or more default types, if an omitted type is not the rightmost type parameter in the type-parameter list, then all type parameters to the right of that type also must be omitted.

Performance Tip 22.2

When appropriate, specify the size of a container class (such as an array class or a stack class) at compile time (possibly through a nontype template parameter). This eliminates the execution-time overhead of using new to create the space dynamically.

Software Engineering Observation 22.3

Specifying the size of a container at compile time avoids the potentially fatal execution-time error if new is unable to obtain the needed memory.

In the exercises, you’ll be asked to use a nontype parameter to create a template for our class Array from Chapter 19. This template will enable Array objects to be instantiated with a specified number of elements of a specified type at compile time, rather than creating space for the Array objects at execution time.

In some cases, it may not be possible to use a particular type with a class template. For example, the Stack template of Fig. 22.2 requires that user-defined types that will be stored in a Stack must provide a default constructor and an assignment operator that
properly copies objects. If a particular user-defined type will not work with our Stack template or requires customized processing, you can define an explicit specialization of the class template for a particular type. Let’s assume we want to create an explicit specialization Stack for Employee objects. To do this, form a new class with the name Stack<Employee> as follows:

```cpp
template<>
class Stack<Employee>
{
    // body of class definition
};
```

The Stack<Employee> explicit specialization is a complete replacement for the Stack class template that is specific to type Employee—it does not use anything from the original class template and can even have different members.

### 22.6 Notes on Templates and Inheritance

Templates and inheritance relate in several ways:

- A class template can be derived from a class-template specialization.
- A class template can be derived from a nontemplate class.
- A class-template specialization can be derived from a class-template specialization.
- A nontemplate class can be derived from a class-template specialization.

### 22.7 Notes on Templates and Friends

We’ve seen that functions and entire classes can be declared as friends of nontemplate classes. With class templates, friendship can be established between a class template and a global function, a member function of another class (possibly a class-template specialization), or even an entire class (possibly a class-template specialization).

Throughout this section, we assume that we’ve defined a class template for a class named X with a single type parameter T, as in:

```cpp
template< typename T >
class X
```

Under this assumption, it’s possible to make a function f1 a friend of every class-template specialization instantiated from the class template for class X. To do so, use a friendship declaration of the form

```cpp
friend void f1();
```

For example, function f1 is a friend of X<double>, X<string> and X<Employee>, etc.

It’s also possible to make a function f2 a friend of only a class-template specialization with the same type argument. To do so, use a friendship declaration of the form

```cpp
friend void f2( X< T > & );
```

For example, if T is a float, function f2( X<float> & ) is a friend of class-template specialization X<float> but not a friend of class-template specification X<string>.

You can declare that a member function of another class is a friend of any class-template specialization generated from the class template. To do so, the friend declaration
must qualify the name of the other class’s member function using the class name and the binary scope resolution operator, as in:

```cpp
friend void A::f3();
```

The declaration makes member function \( f3 \) of class \( A \) a friend of every class-template specialization instantiated from the preceding class template. For example, function \( f3 \) of class \( A \) is a friend of \( X< \text{double} >, X< \text{string} > \) and \( X< \text{Employee} >, \) etc.

As with a global function, another class’s member function can be a friend of only a class-template specialization with the same type argument. A friendship declaration of the form

```cpp
friend void C< T >::f4( X< T > &);
```

for a particular type \( T \) such as \( \text{float} \) makes class \( C \)’s member function

```cpp
C< float >::f4( X< float > &)
```

a friend function of only class-template specialization \( X< \text{float} > \).

In some cases, it’s desirable to make an entire class’s set of member functions friends of a class template. In this case, a friend declaration of the form

```cpp
friend class Y;
```

makes every member function of class \( Y \) a friend of every class-template specialization produced from the class template \( X \).

Finally, it’s possible to make all member functions of one class-template specialization friends of another class-template specialization with the same type argument. For example, a friend declaration of the form:

```cpp
friend class Z< T >;
```

indicates that when a class-template specialization is instantiated with a particular type for \( T \) (such as \( \text{float} \)), all members of class \( Z< \text{float} > \) become friends of class-template specialization \( X< \text{float} > \).

### 22.8 Notes on Templates and static Members

What about static data members? Recall that, with a nontemplate class, one copy of each static data member is shared among all objects of the class, and the static data member must be initialized at global namespace scope.

Each class-template specialization instantiated from a class template has its own copy of each static data member of the class template; all objects of that specialization share that one static data member. In addition, as with static data members of nontemplate classes, static data members of class-template specializations must be defined and, if necessary, initialized at global namespace scope. Each class-template specialization gets its own copy of the class template’s static member functions.

### 22.9 Wrap-Up

This chapter introduced one of C++’s most powerful features—templates. You learned how to use function templates to enable the compiler to produce a set of function-template specializations that represent a group of related overloaded functions. We also discussed
how to overload a function template to create a specialized version of a function that handles a particular data type’s processing in a manner that differs from the other function-template specializations. Next, you learned about class templates and class-template specializations. You saw examples of how to use a class template to create a group of related types that each perform identical processing on different data types. Finally, you learned about some of the relationships among templates, friends, inheritance and static members. In the next chapter, we discuss many of C++’s I/O capabilities and demonstrate several stream manipulators that perform various formatting tasks.

Summary

Section 22.1 Introduction
• Templates enable us to specify a range of related (overloaded) functions—called function-template specializations—or a range of related classes—called class-template specializations.

Section 22.2 Function Templates
• To use function-template specializations, you write a single function-template definition. Based on the argument types provided in calls to this function, C++ generates separate specializations to handle each type of call appropriately.
• All function-template definitions begin with the keyword `template` followed by template parameters enclosed in angle brackets (`<` and `>`) each template parameter that represents a type must be preceded by keyword `class` or `typename`. Keywords `typename` and `class` used to specify function-template parameters mean “any fundamental type or user-defined type.”
• Template-definition template parameters are used to specify the kinds of arguments to the function, the return type of the function and to declare variables in the function.
• As with function parameters, the names of template parameters must be unique inside a template definition. Template parameter names need not be unique across different function templates.

Section 22.3 Overloading Function Templates
• A function template may be overloaded in several ways. We can provide other function templates that specify the same function name but different function parameters. A function template can also be overloaded by providing other nontemplate functions with the same function name, but different function parameters. If both the template and non-template versions match a call, the non-template version will be used.

Section 22.4 Class Templates
• Class templates provide the means for describing a class generically and for instantiating classes that are type-specific versions of this generic class.
• Class templates are called parameterized types; they require type parameters to specify how to customize a generic class template to form a specific class-template specialization.
• To use class-template specializations you write one class template. When you need a new type-specific class, the compiler writes the source code for the class-template specialization.
• A class-template definition looks like a conventional class definition, except that it’s preceded by `template< typename T >` (or `template< class T >`) to indicate this is a class-template definition. Type parameter `T` acts as a placeholder for the type of the class to create. The type `T` is mentioned throughout the class definition and member-function definitions as a generic type name.
• Member-function definitions outside a class template each begin with template<typename T> (or template<class T>). Then, each function definition resembles a conventional function definition, except that the generic data in the class always is listed generically as type parameter T. The binary scope-resolution operator is used with the class-template name to tie each member-function definition to the class template’s scope.

Section 22.5 Nontype Parameters and Default Types for Class Templates
• It’s possible to use nontype parameters in the header of a class or function template.
• You can specify a default type for a type parameter in the type-parameter list.
• An explicit specialization of a class template overrides a class template for a specific type.

Section 22.6 Notes on Templates and Inheritance
• A class template can be derived from a class-template specialization. A class template can be derived from a nontemplate class. A class-template specialization can be derived from a class-template specialization. A nontemplate class can be derived from a class-template specialization.

Section 22.7 Notes on Templates and Friends
• Functions and entire classes can be declared as friends of nontemplate classes. With class templates, friendship arrangements can be declared. Friendship can be established between a class template and a global function, a member function of another class (possibly a class-template specialization) or even an entire class (possibly a class-template specialization).

Section 22.8 Notes on Templates and static Members
• Each class-template specialization has its own copy of each static data member; all objects of that specialization share that static data member. Such data members must be defined and, if necessary, initialized at global namespace scope.
• Each class-template specialization gets a copy of the class template’s static member functions.

Terminology
angle brackets (< and >) 834
class keyword in a template type parameter 834
class template 833
class-template definition 838
class-template specialization 833
default type for a type parameter 844
explicit specialization 845
friend of a template 845
function template 833
function-template definition 834
function-template specialization 833
generic programming 833

macro 834
member function of a class-template specialization 845
nontype template parameter 844
overloading a function template 837
parameterized type 838
template 833
template keyword 834
template parameter 834
type parameter 834
type template parameter 834
typename keyword 834

Self-Review Exercises
22.1 State which of the following are true and which are false. If false, explain why.
a) The template parameters of a function-template definition are used to specify the types of the arguments to the function, to specify the return type of the function and to declare variables within the function.
b) Keywords typename and class as used with a template type parameter specifically mean “any user-defined class type.”
c) A function template can be overloaded by another function template with the same function name.

d) Template parameter names among template definitions must be unique.

e) Each member-function definition outside a class template must begin with a template header.

f) A friend function of a class template must be a function-template specialization.

g) If several class-template specializations are generated from a single class template with a single static data member, each of the class-template specializations shares a single copy of the class template’s static data member.

22.2 Fill in the blanks in each of the following:

a) Templates enable us to specify, with a single code segment, an entire range of related functions called ________, or an entire range of related classes called ________.

b) All function-template definitions begin with the keyword ________, followed by a list of template parameters to the function template enclosed in ________.

c) The related functions generated from a function template all have the same name, so the compiler uses ________ resolution to invoke the proper function.

d) Class templates also are called ________ types.

e) The ________ operator is used with a class-template name to tie each member-function definition to the class template’s scope.

f) As with static data members of nontemplate classes, static data members of class-template specializations must also be defined and, if necessary, initialized at ________ scope.

Answers to Self-Review Exercises

22.1  
a) True. b) False. Keywords typename and class in this context also allow for a type parameter of a fundamental type. c) True. d) False. Template parameter names among function templates need not be unique. e) True. f) False. It could be a nontemplate function. g) False. Each class-template specialization will have its own copy of the static data member.

22.2  
a) function-template specializations, class-template specializations. b) template, angle brackets (< and>). c) overloading. d) parameterized. e) binary scope resolution. f) global namespace.

Exercises

22.3  (Selection Sort Function Template) Write a function template selectionSort (see Appendix F for information on this sorting technique). Write a driver program that inputs, sorts and outputs an int array and a float array.

22.4  (Print Array Range) Overload function template printArray of Fig. 22.1 so that it takes two additional integer arguments, namely int lowSubscript and int highSubscript. A call to this function will print only the designated portion of the array. Validate lowSubscript and highSubscript; if either is out of range or if highSubscript is less than or equal to lowSubscript, the overloaded printArray function should return 0; otherwise, printArray should return the number of elements printed. Then modify main to exercise both versions of printArray on arrays a, b and c (lines 22–24 of Fig. 22.1). Be sure to test all capabilities of both versions of printArray.

22.5  (Function Template Overloading) Overload function template printArray of Fig. 22.1 with a nontemplate version that prints an array of character strings in neat, tabular, column format.

22.6  (Operator Overloading in Templates) Write a simple function template for predicate function isEqualTo that compares its two arguments of the same type with the equality operator (==) and returns true if they are equal and false otherwise. Use this function template in a program that calls isEqualTo only with a variety of fundamental types. Now write a separate version of the program that calls isEqualTo with a user-defined class type, but does not overload the equality operator. What
happens when you attempt to run this program? Now overload the equality operator (with the operator function) operator==. Now what happens when you attempt to run this program?

22.7 (Array Class Template) Use an int template nontype parameter numberOfElements and a type parameter elementType to help create a template for the Array class (Figs. 19.6–19.7) we developed in Chapter 19. This template will enable Array objects to be instantiated with a specified number of elements of a specified element type at compile time.

Write a program with class template Array. The template can instantiate an Array of any element type. Override the template with a specific definition for an Array of float elements (class Array<float>). The driver should demonstrate the instantiation of an Array of int through the template and show that an attempt to instantiate an Array of float uses the definition provided in class Array<float>.

22.8 Distinguish between the terms “function template” and “function-template specialization.”

22.9 Explain which is more like a stencil—a class template or a class-template specialization?

22.10 What’s the relationship between function templates and overloading?

22.11 Why might you choose to use a function template instead of a macro?

22.12 What performance problem can result from using function templates and class templates?

22.13 The compiler performs a matching process to determine which function-template specialization to call when a function is invoked. Under what circumstances does an attempt to make a match result in a compile error?

22.14 Why is it appropriate to refer to a class template as a parameterized type?

22.15 Explain why a C++ program would use the statement

\[
\text{Array< Employee > workerList( 100 );}
\]

22.16 Review your answer to Exercise 22.15. Explain why a C++ program might use the statement

\[
\text{Array< Employee > workerList;}
\]

22.17 Explain the use of the following notation in a C++ program:

\[
\text{template< typename T > Array< T >::Array( int s )}
\]

22.18 Why might you use a nontype parameter with a class template for a container such as an array or stack?

22.19 Suppose that a class template has the header

\[
\text{template< typename T > class Ct1}
\]

Describe the friendship relationships established by placing each of the following friend declarations inside this class template. Identifiers beginning with “f” are functions, identifiers beginning with “C” are classes, identifiers beginning with “Ct” are class templates and T is a template type parameter (i.e., T can represent any fundamental or class type).

a) friend void f1();

b) friend void f2( Ct1< T > &);

c) friend void C2::f3();

d) friend void Ct3< T >::f4( Ct1< T > &);

e) friend class C4;

f) friend class Ct5< T >;

22.20 Suppose that class template Employee has a static data member count. Suppose that three class-template specializations are instantiated from the class template. How many copies of the static data member will exist? How will the use of each be constrained (if at all)?
Consciousness ... does not appear to itself chopped up in bits ... A “river” or a “stream” are the metaphors by which it is most naturally described.
—William James

Objectives
In this chapter you’ll learn:

■ To use C++ object-oriented stream input/output.
■ To format input and output.
■ The stream-I/O class hierarchy.
■ To use stream manipulators.
■ To control justification and padding.
■ To determine the success or failure of input/output operations.
■ To tie output streams to input streams.
23.1 Introduction

The C++ standard libraries provide an extensive set of input/output capabilities. This chapter discusses a range of capabilities sufficient for performing most common I/O operations and overviews the remaining capabilities. We discussed some of these features earlier in the text; now we provide a more complete treatment. Many of the I/O features that we'll discuss are object oriented. This style of I/O makes use of other C++ features, such as references, function overloading and operator overloading.

C++ uses type-safe I/O. Each I/O operation is executed in a manner sensitive to the data type. If an I/O member function has been defined to handle a particular data type, then that member function is called to handle that data type. If there is no match between the type of the actual data and a function for handling that data type, the compiler generates an error. Thus, improper data cannot “sneak” through the system (as can occur in C, allowing for some subtle and bizarre errors).

Users can specify how to perform I/O for objects of user-defined types by overloading the stream insertion operator (<<) and the stream extraction operator (>>). This extensibility is one of C++’s most valuable features.
23.2 Streams

C++ I/O occurs in streams, which are sequences of bytes. In input operations, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection, etc.) to main memory. In output operations, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection, etc.).

An application associates meaning with bytes. The bytes could represent characters, raw data, graphics images, digital speech, digital video or any other information an application may require.

The system I/O mechanisms should transfer bytes from devices to memory (and vice versa) consistently and reliably. Such transfers often involve some mechanical motion, such as the rotation of a disk or a tape, or the typing of keystrokes at a keyboard. The time these transfers take is typically much greater than the time the processor requires to manipulate data internally. Thus, I/O operations require careful planning and tuning to ensure optimal performance.

C++ provides both “low-level” and “high-level” I/O capabilities. Low-level I/O capabilities (i.e., unformatted I/O) specify that some number of bytes should be transferred device-to-memory or memory-to-device. In such transfers, the individual byte is the item of interest. Such low-level capabilities provide high-speed, high-volume transfers but are not particularly convenient.

Programmers generally prefer a higher-level view of I/O (i.e., formatted I/O), in which bytes are grouped into meaningful units, such as integers, floating-point numbers, characters, strings and user-defined types. These type-oriented capabilities are satisfactory for most I/O other than high-volume file processing.

23.2.1 Classic Streams vs. Standard Streams

In the past, the C++ classic stream libraries enabled input and output of chars. Because a char normally occupies one byte, it can represent only a limited set of characters (such as those in the ASCII character set). However, many languages use alphabets that contain more characters than a single-byte char can represent. The ASCII character set does not provide these characters; the Unicode® character set does. Unicode is an extensive inter-
national character set that represents the majority of the world's "commercially viable" languages, mathematical symbols and much more. For more information on Unicode, visit www.unicode.org.

C++ includes the standard stream libraries, which enable developers to build systems capable of performing I/O operations with Unicode characters. For this purpose, C++ includes an additional character type called wchar_t, which can store 2-byte Unicode characters. The C++ standard also redesigned the classic C++ stream classes, which processed only chars, as class templates with separate specializations for processing characters of types char and wchar_t, respectively. We use the char type of class templates throughout this book.

23.2.2 iostream Library Header Files
The C++ iostream library provides hundreds of I/O capabilities. Several header files contain portions of the library interface.

Most C++ programs include the <iostream> header file, which declares basic services required for all stream-I/O operations. The <iostream> header file defines the cin, cout, cerr and clog objects, which correspond to the standard input stream, the standard output stream, the unbuffered standard error stream and the buffered standard error stream, respectively. (cerr and clog are discussed in Section 23.2.3.) Both unformatted- and formatted-I/O services are provided.

The <iomanip> header declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as setw and setprecision. The <fstream> header declares services for user-controlled file processing.

C++ implementations generally contain other I/O-related libraries that provide system-specific capabilities, such as the controlling of special-purpose devices for audio and video I/O.

23.2.3 Stream Input/Output Classes and Objects
The iostream library provides many templates for handling common I/O operations. For example, class template basic_istream supports stream-input operations, class template basic_ostream supports stream-output operations, and class template basic_iostream supports both stream-input and stream-output operations. Each template has a predefined specialization that enables char I/O. In addition, the iostream library provides a set of typedefs that provide aliases for these template specializations. The typedef specifier declares synonyms (aliases) for previously defined data types. Programmers sometimes use typedef to create shorter or more readable type names. For example, the statement

```c++
typedef Card *CardPtr;
```

defines an additional type name, CardPtr, as a synonym for type Card *. Creating a name using typedef does not create a data type; typedef creates only a type name that may be used in the program. Section 10.6 discusses typedef in detail. The typedef istream represents a specialization of basic_istream that enables char input. Similarly, the typedef ostream represents a specialization of basic_ostream that enables char output. Also, the typedef iostream represents a specialization of basic_iostream that enables both char input and output. We use these typedefs throughout this chapter.
### Stream-I/O Template Hierarchy and Operator Overloading

Templates `basic_istream` and `basic_ostream` both derive through single inheritance from base template `basic_ios`.\(^1\) Template `basic_iostream` derives through multiple inheritance from templates `basic_istream` and `basic_ostream`. The UML class diagram of Fig. 23.1 summarizes these inheritance relationships.

![Stream-I/O template hierarchy](image)

---

**Fig. 23.1 | Stream-I/O template hierarchy portion.**

Operator overloading provides a convenient notation for performing input/output. The left-shift operator (`<<`) is overloaded to designate stream output and is referred to as the stream insertion operator. The right-shift operator (`>>`) is overloaded to designate stream input and is referred to as the stream extraction operator. These operators are used with the standard stream objects `cin`, `cout`, `cerr` and `clog` and, commonly, with user-defined stream objects.

**Standard Stream Objects `cin, cout, cerr and clog`**

Predefined object `cin` is an `istream` instance and is said to be “connected to” (or attached to) the standard input device, which usually is the keyboard. The stream extraction operator (`>>`) as used in the following statement causes a value for integer variable `grade` (assuming that `grade` has been declared as an `int` variable) to be input from `cin` to memory:

```cpp
cin >> grade;  // data "flows" in the direction of the arrows
```

The compiler determines the data type of `grade` and selects the appropriate overloaded stream extraction operator. Assuming that `grade` has been declared properly, the stream extraction operator does not require additional type information (as is the case, for example, in C-style I/O). The `>>` operator is overloaded to input data items of fundamental types, strings and pointer values.

The predefined object `cout` is an `ostream` instance and is said to be “connected to” the standard output device, which usually is the display screen. The stream insertion operator (`<<`), as used in the following statement, causes the value of variable `grade` to be output from memory to the standard output device:

```cpp
cout << grade;  // data "flows" in the direction of the arrows
```

---

1. This chapter discusses templates only in the context of the template specializations for `char` I/O.
The compiler determines the data type of \textit{grade} (assuming \textit{grade} has been declared properly) and selects the appropriate stream insertion operator. The \texttt{<<} operator is overloaded to output data items of fundamental types, strings and pointer values.

The predefined object \texttt{cerr} is an \texttt{ostream} instance and is said to be “connected to” the standard error device, normally the screen. Outputs to object \texttt{cerr} are \texttt{unbuffered}, implying that each stream insertion to \texttt{cerr} causes its output to appear immediately—this is appropriate for notifying a user promptly about errors.

The predefined object \texttt{clog} is an instance of the \texttt{ostream} class and is said to be “connected to” the standard error device. Outputs to \texttt{clog} are \texttt{buffered}. This means that each insertion to \texttt{clog} could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed. Buffering is an I/O performance-enhancement technique discussed in operating-systems courses.

\textbf{File-Processing Templates}

C++ file processing uses class templates \texttt{basic\_ifstream} (for file input), \texttt{basic\_ofstream} (for file output) and \texttt{basic\_fstream} (for file input and output). Each class template has a predefined template specialization that enables \texttt{char} I/O. C++ provides a set of \texttt{typedef}s that provide aliases for these template specializations. For example, the \texttt{typedef ifstream} represents a specialization of \texttt{basic\_ifstream} that enables \texttt{char} input from a file. Similarly, \texttt{typedef ofstream} represents a specialization of \texttt{basic\_ofstream} that enables \texttt{char} output to a file. Also, \texttt{typedef fstream} represents a specialization of \texttt{basic\_fstream} that enables \texttt{char} input from, and output to, a file. Template \texttt{basic\_ifstream} inherits from \texttt{basic\_istream}, \texttt{basic\_ofstream} inherits from \texttt{basic\_ostream} and \texttt{basic\_fstream} inherits from \texttt{basic\_iostream}. The UML class diagram of Fig. 23.2 summarizes the various inheritance relationships of the I/O-related classes. The full stream-I/O class hierarchy provides most of the capabilities that you need. Consult the class-library reference for your C++ system for additional file-processing information.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{fig23_02.png}
\caption{Stream-I/O template hierarchy portion showing the main file-processing templates.}
\end{figure}
23.3 Stream Output

Formatted and unformatted output capabilities are provided by \texttt{ostream}. Capabilities include output of standard data types with the stream insertion operator (\texttt{<<}); output of characters via the \texttt{put} member function; unformatted output via the \texttt{write} member function (Section 23.5); output of integers in decimal, octal and hexadecimal formats (Section 23.6.1); output of floating-point values with various precision (Section 23.6.2), with forced decimal points (Section 23.7.1), in scientific notation and in fixed notation (Section 23.7.5); output of data justified in fields of designated widths (Section 23.7.2); output of data in fields padded with specified characters (Section 23.7.3); and output of uppercase letters in scientific notation and hexadecimal notation (Section 23.7.6).

23.3.1 Output of char* Variables

C++ determines data types automatically—an improvement over C. This feature sometimes “gets in the way.” For example, suppose we want to print the address stored in a char* pointer. The \texttt{<<} operator has been overloaded to output a char* as a null-terminated string. To output the address, you can cast the char* to a void* (this can be done to any pointer variable). Figure 23.3 demonstrates printing a char* variable in both string and address formats. The address prints as a hexadecimal (base-16) number, which might differ among computers. To learn more about hexadecimal numbers, read Appendix C. We say more about controlling the bases of numbers in Section 23.6.1 and Section 23.7.4.

~

```
// Fig. 23.3: Fig23_03.cpp
// Printing the address stored in a char* variable.
#include <iostream>
using namespace std;

int main()
{
    const char* const word = "again";

    // display value of char*, then display value of char*
    // static_cast to void*
    cout << "Value of word is: " << word << end1
    << "Value of static_cast< void*>( word ) is: "
    << static_cast< void*>( word ) << end1;
} // end main
```

Value of word is: again
Value of static\_cast\< void\>( word ) is: 00428300

Fig. 23.3 | Printing the address stored in a char* variable.

23.3.2 Character Output Using Member Function put

We can use the \texttt{put} member function to output characters. For example, the statement

\texttt{cout.put( 'A' );}
displays a single character A. Calls to put may be cascaded, as in the statement

```cpp
cout.put('A').put('\n');
```

which outputs the letter A followed by a newline character. As with <<, the preceding statement executes in this manner, because the dot operator (.) associates from left to right, and the put member function returns a reference to the ostream object (cout) that received the put call. The put function also may be called with a numeric expression that represents an ASCII value, as in the following statement

```cpp
cout.put(65);
```

which also outputs A.

### 23.4 Stream Input

Now let’s consider stream input. Formatted and unformatted input capabilities are provided by istream. The stream extraction operator (>>) normally skips white-space characters (such as blanks, tabs, and newlines) in the input stream; later we’ll see how to change this behavior. After each input, the stream extraction operator returns a reference to the stream object that received the extraction message (e.g., cin in the expression cin >> grade). If that reference is used as a condition (e.g., in a while statement’s loop-continuation condition), the stream’s overloaded void * cast operator function is implicitly invoked to convert the reference into a non-null pointer value or the null pointer based on the success or failure of the last input operation. A non-null pointer converts to the bool value true to indicate success and the null pointer converts to the bool value false to indicate failure. When an attempt is made to read past the end of a stream, the stream’s overloaded void * cast operator returns the null pointer to indicate end-of-file.

Each stream object contains a set of state bits used to control the stream’s state (i.e., formatting, setting error states, etc.). These bits are used by the stream’s overloaded void * cast operator to determine whether to return a non-null pointer or the null pointer. Stream extraction causes the stream’s failbit to be set if data of the wrong type is input and causes the stream’s badbit to be set if the operation fails. Section 23.7 and Section 23.8 discuss stream state bits in detail, then show how to test these bits after an I/O operation.

#### 23.4.1 get and getline Member Functions

The get member function with no arguments inputs one character from the designated stream (including white-space characters and other nongraphic characters, such as the key sequence that represents end-of-file) and returns it as the value of the function call. This version of get returns EOF when end-of-file is encountered on the stream.

**Using Member Functions eof, get and put**

Figure 23.4 demonstrates the use of member functions eof and get on input stream cin and member function put on output stream cout. The program first prints the value of cin.eof()—i.e., false (0 on the output)—to show that end-of-file has not occurred on cin. The user enters a line of text and presses Enter followed by end-of-file (<Ctrl>-z on Microsoft Windows systems, <Ctrl>-d on UNIX and Macintosh systems). Line 15 reads each character, which line 16 outputs to cout using member function put. When end-of-file is encountered, the while statement ends, and line 20 displays the value of cin.eof(),
which is now true (1 on the output), to show that end-of-file has been set on cin. This
program uses the version of istream member function get that takes no arguments and
returns the character being input (line 15). Function eof returns true only after the pro-
gram attempts to read past the last character in the stream.

```cpp
// Fig. 23.4: Fig23_04.cpp
// Using member functions get, put and eof.
#include <iostream>
using namespace std;

int main()
{
    int character; // use int, because char cannot represent EOF

    // prompt user to enter line of text
    cout << "Before input, cin.eof() is " << cin.eof() << endl
        << "Enter a sentence followed by end-of-file:" << endl;

    // use get to read each character; use put to display it
    while ((character = cin.get()) != EOF)
    { // display end-of-file character
        cout << "\nEOF in this system is: " << character << endl;
    }

    // display end of file character
    cout << "After input of EOF, cin.eof() is " << cin.eof() << endl;
}
```

Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions
Testing the get and put member functions
^Z
EOF in this system is: -1
After input of EOF, cin.eof() is 1

**Fig. 23.4 | get, put and eof member functions.**

The get member function with a character-reference argument inputs the next char-
acter from the input stream (even if this is a white-space character) and stores it in the character argument. This version of get returns a reference to the istream object for which the get member function is being invoked.

A third version of get takes three arguments—a character array, a size limit and a delimiter (with default value \'\n\'). This version reads characters from the input stream. It either reads one fewer than the specified maximum number of characters and terminates or terminates as soon as the delimiter is read. A null character is inserted to terminate the input string in the character array used as a buffer by the program. The delimiter is not placed in the character array but does remain in the input stream (the delimiter will be the next character read). Thus, the result of a second consecutive get is an empty line, unless the delimiter character is removed from the input stream (possibly with cin.ignore()).
Comparing \texttt{cin} and \texttt{cin.get}

Figure 23.5 compares input using stream extraction with \texttt{cin} (which reads characters until a white-space character is encountered) and input using \texttt{cin.get}. The call to \texttt{cin.get} (line 22) does not specify a delimiter, so the default ‘\n’ character is used.

```cpp
// Fig. 23.5: Fig23_05.cpp
// Contrasting input of a string via cin and cin.get.
#include <iostream>
using namespace std;

int main()
{
    // create two char arrays, each with 80 elements
    const int SIZE = 80;
    char buffer1[ SIZE ];
    char buffer2[ SIZE ];

    // use cin to input characters into buffer1
    cout << "Enter a sentence:" << endl;
    cin >> buffer1;

    // display buffer1 contents
    cout << "The string read with cin was:" << endl
         << buffer1 << endl << endl;

    // use cin.get to input characters into buffer2
    cin.get( buffer2, SIZE );

    // display buffer2 contents
    cout << "The string read with cin.get was:" << endl
         << buffer2 << endl;
}
```

\textbf{Fig. 23.5} | Input of a string using \texttt{cin} with stream extraction contrasted with input using \texttt{cin.get}.

\textbf{Using Member Function getline}

Member function \texttt{getline} operates similarly to the third version of the \texttt{get} member function and inserts a null character after the line in the character array. The \texttt{getline} function removes the delimiter from the stream (i.e., reads the character and discards it), but does not store it in the character array. The program of Fig. 23.6 demonstrates the use of the \texttt{getline} member function to input a line of text (line 13).
23.4.2 istream Member Functions peek, putback and ignore

The `ignore` member function of `istream` reads and discards a designated number of characters (the default is one) or terminates upon encountering a designated delimiter (the default is EOF, which causes `ignore` to skip to the end of the file when reading from a file).

The `putback` member function places the previous character obtained by a `get` from an input stream back into that stream. This function is useful for applications that scan an input stream looking for a field beginning with a specific character. When that character is input, the application returns the character to the stream, so the character can be included in the input data.

The `peek` member function returns the next character from an input stream but does not remove the character from the stream.

23.4.3 Type-Safe I/O

C++ offers type-safe I/O. The `<<` and `>>` operators are overloaded to accept data items of specific types. If unexpected data is processed, various error bits are set, which the user may test to determine whether an I/O operation succeeded or failed. If operator `<<` has not been overloaded for a user-defined type and you attempt to input into or output the contents of an object of that user-defined type, the compiler reports an error. This enables the program to “stay in control.” We discuss these error states in Section 23.8.

23.5 Unformatted I/O Using read, write and gcount

Unformatted input/output is performed using the `read` and `write` member functions of `istream` and `ostream`, respectively. Member function `read` inputs bytes to a character ar-

```cpp
// Fig. 23.6: Fig23_06.cpp
// Inputting characters using cin member function getline.
#include <iostream>
using namespace std;

int main()
{
    const int SIZE = 80;
    char buffer[SIZE]; // create array of 80 characters

    cout << "Enter a sentence:" << endl;
    cin.getline(buffer, SIZE);

    cout << "The sentence entered is:" << endl << buffer << endl;
}
```

Enter a sentence:
Using the getline member function
The sentence entered is:
Using the getline member function

Fig. 23.6 | Inputting character data with `cin` member function `getline`.
ray in memory; member function write outputs bytes from a character array. These bytes are not formatted in any way. They’re input or output as raw bytes. For example, the call

```cpp
char buffer[] = "HAPPY BIRTHDAY";
cout.write( buffer, 10 );
```

outputs the first 10 bytes of buffer (including null characters, if any, that would cause output with cout and « to terminate). The call

```cpp
cout.write( "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10 );
```

displays the first 10 characters of the alphabet.

The read member function inputs a designated number of characters into a character array. If fewer than the designated number of characters are read, failbit is set. Section 23.8 shows how to determine whether failbit has been set. Member function gcount reports the number of characters read by the last input operation.

Figure 23.7 demonstrates istream member functions read and gcount, and ostream member function write. The program inputs 20 characters (from a longer input sequence) into the array buffer with read (line 13), determines the number of characters input with gcount (line 17) and outputs the characters in buffer with write (line 17).

```cpp
1 // Fig. 23.7: Fig23_07.cpp
2 // Unformatted I/O using read, gcount and write.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 80;
9     char buffer[ SIZE ]; // create array of 80 characters
10
11     // use function read to input characters into buffer
12     cout << "Enter a sentence:" << endl;
13     cin.read( buffer, 20 );
14
15     // use functions write and gcount to display buffer characters
16     cout << endl << "The sentence entered was:" << endl;
17     cout.write( buffer, cin.gcount() );
18     cout << endl;
19 } // end main
```

Enter a sentence:
Using the read, write, and gcount member functions
The sentence entered was:
Using the read, write

Fig. 23.7 | Unformatted I/O using the read, gcount and write member functions.

### 23.6 Introduction to Stream Manipulators

C++ provides various stream manipulators that perform formatting tasks. The stream manipulators provide capabilities such as setting field widths, setting precision, setting and
unsetting format state, setting the fill character in fields, flushing streams, inserting a newline into the output stream (and flushing the stream), inserting a null character into the output stream and skipping white space in the input stream. These features are described in the following sections.

23.6.1 Integral Stream Base: dec, oct, hex and setbase

Integers are interpreted normally as decimal (base-10) values. To change the base in which integers are interpreted on a stream, insert the hex manipulator to set the base to hexadecimal (base 16) or insert the oct manipulator to set the base to octal (base 8). Insert the dec manipulator to reset the stream base to decimal. These are all sticky manipulators.

The base of a stream also may be changed by the setbase stream manipulator, which takes one integer argument of 10, 8, or 16 to set the base to decimal, octal or hexadecimal, respectively. Because setbase takes an argument, it’s called a parameterized stream manipulator. Using setbase (or any other parameterized manipulator) requires the inclusion of the <iomanip> header file. The stream base value remains the same until changed explicitly; setbase settings are “sticky.” Figure 23.8 demonstrates stream manipulators hex, oct, dec and setbase.

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int number;

    cout << "Enter a decimal number: ";
    cin >> number;  // input number

    // use hex stream manipulator to show hexadecimal number
    cout << number << " in hexadecimal is: " << hex
        << number << endl;

    // use oct stream manipulator to show octal number
    cout << dec << number << " in octal is: "
        << oct << number << endl;

    // use setbase stream manipulator to show decimal number
    cout << setbase( 10 ) << number << " in decimal is: "
        << number << endl;
}
```

Fig. 23.8 | Stream manipulators hex, oct, dec and setbase.
23.6.2 Floating-Point Precision (precision, setprecision)

We can control the precision of floating-point numbers (i.e., the number of digits to the right of the decimal point) by using either the setprecision stream manipulator or the precision member function of ios_base. A call to either of these sets the precision for all subsequent output operations until the next precision-setting call. A call to member function precision with no argument returns the current precision setting (this is what you need to use so that you can restore the original precision eventually after a “sticky” setting is no longer needed). The program of Fig. 23.9 uses both member function precision (line 22) and the setprecision manipulator (line 31) to print a table that shows the square root of 2, with precision varying from 0 to 9.

```cpp
1 // Fig. 23.9: Fig23_09.cpp
2 // Controlling precision of floating-point values.
3 #include <iostream>
4 #include <iomanip>
5 #include <cmath>
6 #include <omanip>
7 using namespace std;
8
9 int main()
10 {
11   double root2 = sqrt(2.0); // calculate square root of 2
12   int places; // precision, vary from 0-9
13   
14   cout << "Square root of 2 with precisions 0-9." << endl
15   << "Precision set by ios_base member function "
16   << "precision:" << endl;
17   cout << fixed; // use fixed-point notation
18   // display square root using ios_base function precision
19   for ( places = 0; places <= 9; places++ )
20   {
21       cout.precision( places );
22       cout << root2 << endl;
23   } // end for
24   cout << "\nPrecision set by stream manipulator "
25   << "setprecision:" << endl;
26   // set precision for each digit, then display square root
27   for ( places = 0; places <= 9; places++ )
28       cout << setprecision( places ) << root2 << endl;
29 } // end main
```

Square root of 2 with precisions 0-9.
Precision set by ios_base member function precision:

1
1.4
1.41
1.414

Fig. 23.9 | Precision of floating-point values. (Part 1 of 2.)
23.6.3 Field Width (width, setw)

The width member function (of base class ios_base) sets the field width (i.e., the number of character positions in which a value should be output or the maximum number of characters that should be input) and returns the previous width. If values output are narrower than the field width, fill characters are inserted as padding. A value wider than the designated width will not be truncated—the full number will be printed. The width function with no argument returns the current setting.

Common Programming Error 23.1

The width setting applies only for the next insertion or extraction (i.e., the width setting is not "sticky"); afterward, the width is set implicitly to 0 (i.e., input and output will be performed with default settings). Assuming that the width setting applies to all subsequent outputs is a logic error.

Common Programming Error 23.2

When a field is not sufficiently wide to handle outputs, the outputs print as wide as necessary, which can yield confusing outputs.

Figure 23.10 demonstrates the use of the width member function on both input and output. On input into a char array, a maximum of one fewer characters than the width will be read, because provision is made for the null character to be placed in the input string. Remember that stream extraction terminates when nonleading white space is encountered. The setw stream manipulator also may be used to set the field width.

```
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

Fig. 23.9 | Precision of floating-point values. (Part 2 of 2.)
```

Fig. 23.10 | width member function of class ios_base. (Part 1 of 2.)
Chapter 23  Stream Input/Output

Note: When prompted for input in Fig. 23.10, the user should enter a line of text and press Enter followed by end-of-file (\(^\text{Ctrl}\)-z on Microsoft Windows systems, \(^\text{Ctrl}\)-d on UNIX and Macintosh systems).

23.6.4 User-Defined Output Stream Manipulators

You can create your own stream manipulators. Figure 23.11 shows the creation and use of new nonparameterized stream manipulators bell, carriageReturn, tab and endLine. For output stream manipulators, the return type and parameter must be of type ostream&. When line 35 inserts the endLine manipulator in the output stream, function endLine is called and line 29 outputs the escape sequence \n and the flush manipulator to the standard output stream cout. Similarly, when lines 35–44 insert the manipulators tab, bell and carriageReturn in the output stream, their corresponding functions—tab, bell and carriageReturn—are called, which in turn output various escape sequences.

2. You also may create your own parameterized stream manipulators. This concept is beyond the scope of this book.
23.6 Introduction to Stream Manipulators

```cpp
// Fig. 23.11: Fig23_11.cpp
// Creating and testing user-defined, nonparameterized
// stream manipulators.
#include <iostream>
using namespace std;

// bell manipulator (using escape sequence \a)
ostream& bell( ostream& output )
{
    return output << '\a'; // issue system beep
} // end bell manipulator

// carriageReturn manipulator (using escape sequence \r)
ostream& carriageReturn( ostream& output )
{
    return output << '\r'; // issue carriage return
} // end carriageReturn manipulator

// tab manipulator (using escape sequence \t)
ostream& tab( ostream& output )
{
    return output << '\t'; // issue tab
} // end tab manipulator

// endLine manipulator (using escape sequence \n and member
// function flush)
ostream& endLine( ostream& output )
{
    return output << '\n' << flush; // issue endl-like end of line
} // end endLine manipulator

int main()
{
    // use tab and endLine manipulators
    cout << "Testing the tab manipulator:" << endLine
         << 'a' << tab << 'b' << tab << 'c' << endLine;
    cout << "Testing the carriageReturn and bell manipulators:
         " << endLine << "..........";
    cout << bell; // use bell manipulator
    // use carriageReturn and endLine manipulators
    cout << carriageReturn << "-----" << endLine;
} // end main
```

Testing the tab manipulator:
```
 a   b   c
```
Testing the carriageReturn and bell manipulators:
```
-----.....
```
23.7 Stream Format States and Stream Manipulators

Various stream manipulators can be used to specify the kinds of formatting to be performed during stream-I/O operations. Stream manipulators control the output’s format settings. Figure 23.12 lists each stream manipulator that controls a given stream’s format state. All these manipulators belong to class `ios_base`. We show examples of most of these stream manipulators in the next several sections.

<table>
<thead>
<tr>
<th>Stream manipulator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>skipws</code></td>
<td>Skip white-space characters on an input stream. This setting is reset with stream manipulator <code>noskipws</code>.</td>
</tr>
<tr>
<td><code>left</code></td>
<td>Left justify output in a field. Padding characters appear to the right if necessary.</td>
</tr>
<tr>
<td><code>right</code></td>
<td>Right justify output in a field. Padding characters appear to the left if necessary.</td>
</tr>
<tr>
<td><code>internal</code></td>
<td>Indicate that a number’s sign should be left justified in a field and a number’s magnitude should be right justified in that same field (i.e., padding characters appear between the sign and the number).</td>
</tr>
<tr>
<td><code>dec</code></td>
<td>Specify that integers should be treated as decimal (base 10) values.</td>
</tr>
<tr>
<td><code>oct</code></td>
<td>Specify that integers should be treated as octal (base 8) values.</td>
</tr>
<tr>
<td><code>hex</code></td>
<td>Specify that integers should be treated as hexadecimal (base 16) values.</td>
</tr>
<tr>
<td><code>showbase</code></td>
<td>Specify that the base of a number is to be output ahead of the number (a leading 0 for octals; a leading 0x or 0X for hexadecimals). This setting is reset with stream manipulator <code>noshowbase</code>.</td>
</tr>
<tr>
<td><code>showpoint</code></td>
<td>Specify that floating-point numbers should be output with a decimal point. This is used normally with <code>fixed</code> to guarantee a certain number of digits to the right of the decimal point, even if they’re zeros. This setting is reset with stream manipulator <code>noshowpoint</code>.</td>
</tr>
<tr>
<td><code>uppercase</code></td>
<td>Specify that uppercase letters (i.e., X and A through F) should be used in a hexadecimal integer and that uppercase E should be used when representing a floating-point value in scientific notation. This setting is reset with stream manipulator <code>nouppercase</code>.</td>
</tr>
<tr>
<td><code>showpos</code></td>
<td>Specify that positive numbers should be preceded by a plus sign (+). This setting is reset with stream manipulator <code>noshowpos</code>.</td>
</tr>
<tr>
<td><code>scientific</code></td>
<td>Specify output of a floating-point value in scientific notation.</td>
</tr>
<tr>
<td><code>fixed</code></td>
<td>Specify output of a floating-point value in fixed-point notation with a specific number of digits to the right of the decimal point.</td>
</tr>
</tbody>
</table>

Fig. 23.12 | Format state stream manipulators from `<iostream>`.  

23.7.1 Trailing Zeros and Decimal Points (showpoint)

Stream manipulator `showpoint` forces a floating-point number to be output with its decimal point and trailing zeros. For example, the floating-point value 79.0 prints as 79 with-
out using showpoint and prints as 79.000000 (or as many trailing zeros as are specified by the current precision) using showpoint. To reset the showpoint setting, output the stream manipulator noshowpoint. The program in Fig. 23.13 shows how to use stream manipulator showpoint to control the printing of trailing zeros and decimal points for floating-point values. Recall that the default precision of a floating-point number is 6. When neither the fixed nor the scientific stream manipulator is used, the precision represents the number of significant digits to display (i.e., the total number of digits to display), not the number of digits to display after decimal point.

```cpp
// Fig. 23.13: Fig23_13.cpp
// Controlling the printing of trailing zeros and decimal points in floating-point values.
#include <iostream>
using namespace std;

int main()
{
    // display double values with default stream format
    cout << "Before using showpoint" << endl
         << "9.9900 prints as: " << 9.9900 << endl
         << "9.9000 prints as: " << 9.9000 << endl
         << "9.0000 prints as: " << 9.0000 << endl;

    // display double value after showpoint
    cout << showpoint
         << "After using showpoint" << endl
         << "9.9900 prints as: " << 9.9900 << endl
         << "9.9000 prints as: " << 9.9000 << endl
         << "9.0000 prints as: " << 9.0000 << endl;
} // end main
```

Before using showpoint
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9

After using showpoint
9.9900 prints as: 9.990000
9.9000 prints as: 9.900000
9.0000 prints as: 9.000000

Fig. 23.13 | Controlling the printing of trailing zeros and decimal points in floating-point values.

### 23.7.2 Justification (left, right and internal)

Stream manipulators `left` and `right` enable fields to be left justified with padding characters to the right or right justified with padding characters to the left, respectively. The padding character is specified by the `fill` member function or the `setfill` parameterized stream manipulator (which we discuss in Section 23.7.3). Figure 23.14 uses the `setw`, `left` and `right` manipulators to left justify and right justify integer data in a field.
Stream manipulator `internal` indicates that a number’s sign (or base when using stream manipulator `showbase`) should be left justified within a field, that the number’s magnitude should be right justified and that intervening spaces should be padded with the fill character. Figure 23.15 shows the `internal` stream manipulator specifying internal spacing (line 10). Note that `showpos` forces the plus sign to print (line 10). To reset the `showpos` setting, output the stream manipulator `noshowpos`.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int x = 12345;

    // display x right justified (default)
    cout << "Default is right justified:" << endl
         << setw(10) << x;

    // use left manipulator to display x left justified
    cout << "\n\nUse std::left to left justify x:\n"     
         << left << setw(10) << x;

    // use right manipulator to display x right justified
    cout << "\n\nUse std::right to right justify x:\n"     
         << right << setw(10) << x << endl;
}
```
23.7 Stream Format States and Stream Manipulators

23.7.3 Padding (fill, setfill)

The fill member function specifies the fill character to be used with justified fields; if no value is specified, spaces are used for padding. The fill function returns the prior padding character. The setfill manipulator also sets the padding character. Figure 23.16 demonstrates using member function fill (line 30) and stream manipulator setfill (lines 34 and 37) to set the fill character.

```
// Fig. 23.16: Fig23_16.cpp
// Using member function fill and stream manipulator setfill to change
// the padding character for fields larger than the printed value.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int x = 10000;

    // display x
    cout << x << " printed as int right and left justified\n"
         << "and as hex with internal justification.\n"
         << "Using the default pad character (space):" << endl;

    // display x with base
    cout << showbase << setw(10) << x << endl;

    // display x with left justification
    cout << left << setw(10) << x << endl;

    // display x as hex with internal justification
    cout << internal << setw(10) << hex << x << endl << endl;

    cout << "Using various padding characters:" << endl;

    // display x using padded characters (right justification)
    cout << right;
    cout.fill(‘*’);
    cout << setw(10) << dec << x << endl;

    // display x using padded characters (left justification)
    cout << left << setw(10) << setfill(‘%’) << x << endl;
```

Fig. 23.15 | Printing an integer with internal spacing and plus sign. (Part 2 of 2.)

Fig. 23.16 | Using member function fill and stream manipulator setfill to change the padding character for fields larger than the values being printed. (Part 1 of 2.)
Chapter 23  Stream Input/Output

23.7.4 Integral Stream Base (dec, oct, hex, showbase)

C++ provides stream manipulators `dec`, `hex` and `oct` to specify that integers are to be displayed as decimal, hexadecimal and octal values, respectively. Stream insertions default to decimal if none of these manipulators is used. With stream extraction, integers prefixed with `0` (zero) are treated as octal values, integers prefixed with `0x` or `0X` are treated as hexadecimal values, and all other integers are treated as decimal values. Once a particular base is specified for a stream, all integers on that stream are processed using that base until a different base is specified or until the program terminates.

Stream manipulator `showbase` forces the base of an integral value to be output. Decimal numbers are output by default, octal numbers are output with a leading `0`, and hexadecimal numbers are output with either a leading `0x` or a leading `0X` (as we discuss in Section 23.7.6, stream manipulator `uppercase` determines which option is chosen). Figure 23.17 demonstrates the use of stream manipulator `showbase` to force an integer to print in decimal, octal and hexadecimal formats. To reset the `showbase` setting, output the stream manipulator `noshowbase`.

```cpp
// display x using padded characters (internal justification)
std::cout << internal << std::setw(10) << std::setfill('^') << std::hex << x << std::endl;

10000 printed as int right and left justified and as hex with internal justification.
Using the default pad character (space):
10000
0x 2710
Using various padding characters:
*****10000
10000%%%%%
0x^^^^2710
```

**Fig. 23.16**  Using member function `fill` and stream manipulator `setfill` to change the padding character for fields larger than the values being printed. (Part 2 of 2.)

```cpp
#include <iostream>

using namespace std;

int main()
{
    int x = 100;
    // use showbase to show number base
    std::cout << "Printing integers preceded by their base:" << std::endl
                << "showbase;"
```

**Fig. 23.17**  Stream manipulator `showbase`. (Part I of 2.)
23.7.5 Floating-Point Numbers; Scientific and Fixed Notation

(scientific, fixed)

Stream manipulators scientific and fixed control the output format of floating-point numbers. Stream manipulator scientific forces the output of a floating-point number to display in scientific format. Stream manipulator fixed forces a floating-point number to display a specific number of digits (as specified by member function precision or stream manipulator setprecision) to the right of the decimal point. Without using another manipulator, the floating-point-number value determines the output format.

Figure 23.18 demonstrates displaying floating-point numbers in fixed and scientific formats using stream manipulators scientific (line 18) and fixed (line 22). The exponent format in scientific notation might differ across different compilers.
23.7.6 Uppercase/Lowercase Control (uppercase)

Stream manipulator uppercase outputs an uppercase X or E with hexadecimal-integer values or with scientific notation floating-point values, respectively (Fig. 23.19). Using stream manipulator uppercase also causes all letters in a hexadecimal value to be uppercase. By default, the letters for hexadecimal values and the exponents in scientific notation floating-point values appear in lowercase. To reset the uppercase setting, output the stream manipulator nouppercase.

```cpp
// Fig. 23.19: Fig23_19.cpp
// Stream manipulator uppercase.
#include <iostream>
using namespace std;

int main()
{
    cout << "Printing uppercase letters in scientific"
        << " notation exponents and hexadecimal values:" << endl;
    cout << uppercase << 4.345e10 << endl
        << hex << showbase << 123456789 << endl;
} // end main
```

**Printing uppercase letters in scientific notation exponents and hexadecimal values:**

4.345E+010
0X75BCD15

23.7.7 Specifying Boolean Format (boolalpha)

C++ provides data type bool, whose values may be false or true, as a preferred alternative to the old style of using 0 to indicate false and nonzero to indicate true. A bool variable outputs as 0 or 1 by default. However, we can use stream manipulator boolalpha to set the output stream to display bool values as the strings "true" and "false". Use stream manipulator noboolalpha to set the output stream to display bool values as integers (i.e., the default setting). The program of Fig. 23.20 demonstrates these stream manipulators.
Line 11 displays the bool value, which line 8 sets to true, as an integer. Line 15 uses manipulator boolalpha to display the bool value as a string. Lines 18–19 then change the bool’s value and use manipulator noboolalpha, so line 22 can display the bool value as an integer. Line 26 uses manipulator boolalpha to display the bool value as a string. Both boolalpha and noboolalpha are “sticky” settings.

**Good Programming Practice 23.1**

Displaying bool values as true or false, rather than nonzero or 0, respectively, makes program outputs clearer.

```cpp
// Fig. 23.20: Fig23_20.cpp
// Demonstrating stream manipulators boolalpha and noboolalpha.
#include <iostream>
using namespace std;

int main()
{
    bool booleanValue = true;

    // display default true booleanValue
    cout << "booleanValue is " << booleanValue << endl;

    // display booleanValue after using boolalpha
    cout << "booleanValue (after using boolalpha) is "
         << boolalpha << booleanValue << endl;

    cout << "switch booleanValue and use noboolalpha" << endl;
    booleanValue = false; // change booleanValue
    cout << noboolalpha << booleanValue << endl; // use noboolalpha

    // display default false booleanValue after using noboolalpha
    cout << "booleanValue is " << booleanValue << endl;

    // display booleanValue after using boolalpha again
    cout << "booleanValue (after using boolalpha) is "
         << boolalpha << booleanValue << endl;
} // end main
```

Fig. 23.20 | Stream manipulators boolalpha and noboolalpha.

### 23.7.8 Setting and Resetting the Format State via Member Function flags

Throughout Section 23.7, we’ve been using stream manipulators to change output format characteristics. We now discuss how to return an output stream’s format to its default state.
after having applied several manipulations. Member function `flags` without an argument returns the current format settings as a `fmtflags` data type (of class `ios_base`), which represents the format state. Member function `flags` with a `fmtflags` argument sets the format state as specified by the argument and returns the prior state settings. The initial settings of the value that `flags` returns might differ across several systems. The program of Fig. 23.21 uses member function `flags` to save the stream’s original format state (line 17), then restore the original format settings (line 25).

```cpp
#include <iostream>
using namespace std;

int main()
{
    int integerValue = 1000;
    double doubleValue = 0.0947628;

    // display flags value, int and double values (original format)
    cout << "The value of the flags variable is: " << cout.flags()
        << "\nPrint int and double in original format:\n" << integerValue << '-' << doubleValue << endl;

    // use cout flags function to save original format
    ios_base::fmtflags originalFormat = cout.flags();
    cout << showbase << oct << scientific; // change format

    cout.flags( originalFormat ); // restore format

    // display flags value, int and double values (new format)
    cout << "The value of the flags variable is: " << cout.flags()
        << "\nPrint int and double in a new format:\n" << integerValue << '-' << doubleValue << endl;

    // display flags value, int and double values (original format again)
    cout.flags();
    cout << showbase << oct << scientific; // change format
    cout.flags( originalFormat ); // restore format

    // display flags value, int and double values (original format again)
    cout << "The restored value of the flags variable is: "
        << cout.flags()
        << "\nPrint values in original format again:\n" << integerValue << '-' << doubleValue << endl;

    return 0;
}
```

The value of the flags variable is: 513
Print int and double in original format:
1000 0.0947628

The value of the flags variable is: 012011
Print int and double in a new format:
01750 9.476280e-002

The restored value of the flags variable is: 513
Print values in original format again:
1000 0.0947628

Fig. 23.21  |  flags member function.
23.8 Stream Error States

The state of a stream may be tested through bits in class `ios_base`. In a moment, we show how to test these bits, in the example of Fig. 23.22.

The `eofbit` is set for an input stream after end-of-file is encountered. A program can use member function `eof` to determine whether end-of-file has been encountered on a stream after an attempt to extract data beyond the end of the stream. The call

```
cin.eof()
```

returns true if end-of-file has been encountered on `cin` and `false` otherwise.

The `failbit` is set for a stream when a format error occurs on the stream and no characters are input (e.g., when you attempt to read a number and the user enters a string). When such an error occurs, the characters are not lost. The `fail` member function reports whether a stream operation has failed. Usually, recovering from such errors is possible.

---

```cpp
// Fig. 23.22: Fig23_22.cpp
// Testing error states.
#include <iostream>
using namespace std;

int main()
{
    int integerValue;
    // display results of cin functions
    cout << "Before a bad input operation:"
        << "\ncin.rdstate(): " << cin.rdstate()
        << "\n    cin.eof(): " << cin.eof()
        << "\n   cin.fail(): " << cin.fail()
        << "\n   cin.bad(): " << cin.bad()
        << "\n   cin.good(): " << cin.good()
        << "\n\nExpects an integer, but enter a character: ";
    cin >> integerValue; // enter character value
    cout << endl;
    // display results of cin functions after bad input
    cout << "After a bad input operation:"
        << "\ncin.rdstate(): " << cin.rdstate()
        << "\n    cin.eof(): " << cin.eof()
        << "\n   cin.fail(): " << cin.fail()
        << "\n   cin.bad(): " << cin.bad()
        << "\n   cin.good(): " << cin.good()
        << endl << endl;
    cin.clear(); // clear stream
    // display results of cin functions after clearing cin
    cout << "After cin.clear()"  
        << "\ncin.fail(): " << cin.fail()
        << "\ncin.good(): " << cin.good() << endl;
} // end main
```

---

Fig. 23.22 | Testing error states. (Part 1 of 2.)
Chapter 23 Stream Input/Output

The badbit is set for a stream when an error occurs that results in the loss of data. The bad member function reports whether a stream operation failed. Generally, such serious failures are nonrecoverable.

The goodbit is set for a stream if none of the bits eofbit, failbit or badbit is set for the stream. The good member function returns true if the bad, fail and eof functions would all return false. I/O operations should be performed only on “good” streams.

The rdstate member function returns the stream’s error state. Calling cout.rdstate, for example, would return the stream’s state, which then could be tested by a switch statement that examines eofbit, badbit, failbit and goodbit. The preferred means of testing the state of a stream is to use member functions eof, bad, fail and good—using these functions does not require you to be familiar with particular status bits.

The clear member function is used to restore a stream’s state to “good,” so that I/O may proceed on that stream. The default argument for clear is goodbit, so the statement

```
cin.clear();
```

clears cin and sets goodbit for the stream. The statement

```
cin.clear( ios::failbit )
```

sets the failbit. You might want to do this when performing input on cin with a user-defined type and encountering a problem. The name clear might seem inappropriate in this context, but it’s correct.

The program of Fig. 23.22 demonstrates member functions rdstate, eof, fail, bad, good and clear. [Note: The actual values output may differ across different compilers.]

The operator! member function of basic_ios returns true if the badbit is set, the failbit is set or both are set. The operator void * member function returns false (0) if the badbit is set, the failbit is set or both are set. These functions are useful in file processing when a true/false condition is being tested under the control of a selection statement or repetition statement.
23.9 Tying an Output Stream to an Input Stream

Interactive applications generally involve an istream for input and an ostream for output. When a prompting message appears on the screen, the user responds by entering the appropriate data. Obviously, the prompt needs to appear before the input operation proceeds. With output buffering, outputs appear only when the buffer fills, when outputs are flushed explicitly by the program or automatically at the end of the program. C++ provides member function tie to synchronize (i.e., “tie together”) the operation of an istream and an ostream to ensure that outputs appear before their subsequent inputs. The call

```
cin.tie( &cout );
```

ties cout (an ostream) to cin (an istream). Actually, this particular call is redundant, because C++ performs this operation automatically to create a user's standard input/output environment. However, the user would tie other istream/ostream pairs explicitly. To untie an input stream, inputStream, from an output stream, use the call

```
inputStream.tie( 0 );
```

23.10 Wrap-Up

This chapter summarized how C++ performs input/output using streams. You learned about the stream-I/O classes and objects, as well as the stream I/O template class hierarchy. We discussed ostream’s formatted and unformatted output capabilities performed by the put and write functions. You saw examples using istream’s formatted and unformatted input capabilities performed by the eof, get, getline, peek, putback, ignore and read functions. Next, we discussed stream manipulators and member functions that perform formatting tasks—dec, oct, hex and setbase for displaying integers; precision and set-precision for controlling floating-point precision; and width and setw for setting field width. You also learned additional formatting istream manipulators and member functions—showpoint for displaying decimal point and trailing zeros; left, right and internal for justification; fill and setfill for padding; scientific and fixed for displaying floating-point numbers in scientific and fixed notation; uppercase for uppercase/lowercase control; boolalpha for specifying boolean format; and flags and fmtflags for resetting the format state.

In the next chapter, we introduce exception handling, which allows you to deal with certain problems that may occur during a program’s execution. We demonstrate basic exception-handling techniques that often permit a program to continue executing as if no problem had been encountered. We also present several classes that the C++ Standard Library provides for handling exceptions.

Summary

Section 23.1 Introduction
- I/O operations are performed in a manner sensitive to the type of the data.

Section 23.2 Streams
- C++ I/O occurs in streams. A stream is a sequence of bytes.
I/O mechanisms move bytes from devices to memory and vice versa efficiently and reliably.

C++ provides “low-level” and “high-level” I/O capabilities. Low-level I/O capabilities specify that bytes should be transferred device-to-memory or memory-to-device. High-level I/O is performed with bytes grouped into meaningful units such as integers, strings and user-defined types.

C++ provides both unformatted-I/O and formatted-I/O operations. Unformatted-I/O transfers are fast, but process raw data that is difficult for people to use. Formatted I/O processes data in meaningful units, but requires extra processing time that can degrade the performance.

The `<iostream>` header file declares all stream-I/O operations.

The `<iomanip>` header declares the parameterized stream manipulators.

The `<fstream>` header declares file-processing operations.

The `basic_istream` template supports stream-input operations.

The `basic_ostream` template supports stream-output operations.

The `basic_iostream` template supports both stream-input and stream-output operations.

Templates `basic_istream` and the `basic_ostream` each derive from the `basic_ios` template.

Template `basic_iostream` derives from both the `basic_istream` and `basic_ostream` templates.

The `iostream` object `cin` is tied to the standard input device, normally the keyboard.

The `ostream` object `cout` is tied to the standard output device, normally the screen.

The `ostream` object `cerr` is tied to the standard error device, normally the screen. Outputs to `cerr` are unbuffered; each insertion to `cerr` appears immediately.

The `ostream` object `clog` is tied to the standard error device, normally the screen. Outputs to `clog` are buffered.

The C++ compiler determines data types automatically for input and output.

Section 23.3 Stream Output
• Addresses are displayed in hexadecimal format by default.
• To print the address in a pointer variable, cast the pointer to `void *`.
• Member function `put` outputs one character. Calls to `put` may be cascaded.

Section 23.4 Stream Input
• Stream input is performed with the stream extraction operator `>>`, which automatically skips white-space characters in the input stream and returns `false` after end-of-file is encountered.
• Stream extraction causes `failbit` to be set for improper input and `badbit` to be set if the operation fails.
• A series of values can be input using the stream extraction operation in a `while` loop header. The extraction returns 0 when end-of-file is encountered or an error occurs.
• The `get` member function with no arguments inputs one character and returns the character; `EOF` is returned if end-of-file is encountered on the stream.
• Member function `get` with a character-reference argument inputs the next character from the input stream and stores it in the character argument. This version of `get` returns a reference to the `istream` object for which the `get` member function is being invoked.
• Member function `get` with three arguments—a character array, a size limit and a delimiter (with default value newline)—reads characters from the input stream up to a maximum of limit – 1 characters, or until the delimiter is read. The input string is terminated with a null character. The delimiter is not placed in the character array but remains in the input stream.
• The `getline` member function operates like the three-argument `get` member function. The `getline` function removes the delimiter from the input stream but does not store it in the string.
• Member function `ignore` skips the specified number of characters (the default is 1) in the input stream; it terminates if the specified delimiter is encountered (the default delimiter is `EOF`).

• The `putback` member function places the previous character obtained by a `get` on a stream back into that stream.

• The `peek` member function returns the next character from an input stream but does not extract (remove) the character from the stream.

• C++ offers type-safe I/O. If unexpected data is processed by the `<<` and `>>` operators, various error bits are set, which can be tested to determine whether an I/O operation succeeded or failed. If operator `<<` has not been overloaded for a user-defined type, a compiler error is reported.

**Section 23.5 Unformatted I/O Using `read`, `write` and `gcount`**

• Unformatted I/O is performed with member functions `read` and `write`. These input or output bytes to or from memory, beginning at a designated memory address.

• The `gcount` member function returns the number of characters input by the previous `read` operation on that stream.

• Member function `read` inputs a specified number of characters into a character array. `failbit` is set if fewer than the specified number of characters are read.

**Section 23.6 Introduction to Stream Manipulators**

• To change the base in which integers output, use the manipulator `hex` to set the base to hexadecimal (base 16) or `oct` to set the base to octal (base 8). Use manipulator `dec` to reset the base to decimal. The base remains the same until changed explicitly.

• The parameterized stream manipulator `setbase` also sets the base for integer output. `setbase` takes one integer argument of 10, 8 or 16 to set the base.

• Floating-point precision can be controlled with the `setprecision` stream manipulator or the `precision` member function. Both set the precision for all subsequent output operations until the next precision-setting call. The `precision` member function with no argument returns the current precision value.

• Parameterized manipulators require the inclusion of the `<iomanip>` header file.

• Member function `width` sets the field width and returns the previous width. Values narrower than the field are padded with fill characters. The field-width setting applies only for the next insertion or extraction; the field width is set to 0 implicitly (subsequent values will be output as large as necessary). Values wider than a field are printed in their entirety. Function `width` with no argument returns the current width setting. Manipulator `setw` also sets the width.

• For input, the `setw` stream manipulator establishes a maximum string size; if a larger string is entered, the larger line is broken into pieces no larger than the designated size.

• You can create your own stream manipulators.

**Section 23.7 Stream Format States and Stream Manipulators**

• Stream manipulator `showpoint` forces a floating-point number to be output with a decimal point and with the number of significant digits specified by the precision.

• Stream manipulators `left` and `right` cause fields to be left justified with padding characters to the right or right justified with padding characters to the left.

• Stream manipulator `internal` indicates that a number’s sign (or base when using stream manipulator `showbase`) should be left justified within a field, its magnitude should be right justified and intervening spaces should be padded with the fill character.
• Member function fill specifies the fill character to be used with stream manipulators left, right and internal (space is the default); the prior padding character is returned. Stream manipulator setfill also sets the fill character.

• Stream manipulators oct, hex and dec specify that integers are to be treated as octal, hexadecimal or decimal values, respectively. Integer output defaults to decimal if none of these bits is set; stream extractions process the data in the form the data is supplied.

• Stream manipulator showbase forces the base of an integral value to be output.

• Stream manipulator scientific is used to output a floating-point number in scientific format. Stream manipulator fixed is used to output a floating-point number with the precision specified by the precision member function.

• Stream manipulator uppercase outputs an uppercase X or E for hexadecimal integers and scientific notation floating-point values, respectively. Hexadecimal values appear in all uppercase.

• Member function flags with no argument returns the long value of the current format state settings. Function flags with a long argument sets the format state specified by the argument.

Section 23.8 Stream Error States
• The state of a stream may be tested through bits in class ios_base.

• The eofbit is set for an input stream after end-of-file is encountered during an input operation. The eof member function reports whether the eofbit has been set.

• A stream’s failbit is set when a format error occurs. The fail member function reports whether a stream operation has failed; it’s normally possible to recover from such errors.

• A stream’s badbit is set when an error occurs that results in data loss. Member function bad reports whether such a stream operation failed. Such serious failures are normally nonrecoverable.

• The good member function returns true if the bad, fail and eof functions would all return false. I/O operations should be performed only on “good” streams.

• The rdstate member function returns the error state of the stream.

• Member function clear restores a stream’s state to “good,” so that I/O may proceed.

Section 23.9 Tying an Output Stream to an Input Stream
• C++ provides the tie member function to synchronize istream and ostream operations to ensure that outputs appear before subsequent inputs.

Terminology
bad member function of basic_ios 878
clear member function of basic_ios 878
def stream manipulator 863
eof member function of basic_ios 878
eofbit 877
fail member function of basic_ios 878
failbit 858
fill character 865
fill member function of basic_ios 871
fixed stream manipulator 873
flags member function of ios_base 876
fmtflags 876
format state 876
Self-Review Exercises

23.1 Answer each of the following:

a) Input/output in C++ occurs as _______ of bytes.
b) The stream manipulators that format justification are ________, ________ and ________.

c) Member function ________ can be used to set and reset format state.
d) Most C++ programs that do I/O should include the ________ header file that contains the declarations required for all stream-I/O operations.

e) When using parameterized manipulators, the header file ________ must be included.
f) Header file ________ contains the declarations required for file processing.
g) The ostream member function ________ is used to perform unformatted output.

h) Input operations are supported by class ________.

i) Standard error stream outputs are directed to the stream objects ________ or ________.
j) Output operations are supported by class ________.

k) The symbol for the stream insertion operator is ________.

l) The four objects that correspond to the standard devices on the system include ________, ________, ________, and ________.

m) The symbol for the stream extraction operator is ________.

n) The stream manipulators ________, ________, and ________ specify that integers should be displayed in octal, hexadecimal and decimal formats, respectively.

o) The ________ stream manipulator causes positive numbers to display with a plus sign.
23.2 State whether the following are true or false. If the answer is false, explain why.

a) The stream member function flags with a long argument sets the flags state variable to its argument and returns its previous value.

b) The stream insertion operator << and the stream extraction operator >> are overloaded to handle all standard data types—including strings and memory addresses (stream insertion only)—and all user-defined data types.

c) The stream member function flags with no arguments resets the stream’s format state.

d) The stream extraction operator >> can be overloaded with an operator function that takes an ostream reference and a reference to a user-defined type as arguments and returns an ostream reference.

e) The stream insertion operator << can be overloaded with an operator function that takes an ostream reference and a reference to a user-defined type as arguments and returns an ostream reference.

f) Input with the stream extraction operator >> always skips leading white-space characters in the input stream, by default.

g) The stream member function rdstate returns the current state of the stream.

h) The cout stream normally is connected to the display screen.

i) The stream member function good returns true if the bad, fail and eof member functions all return false.

j) The cin stream normally is connected to the display screen.

k) If a nonrecoverable error occurs during a stream operation, the bad member function will return true.

l) Output to cerr is unbuffered and output to clog is buffered.

m) Stream manipulator showpoint forces floating-point values to print with the default six digits of precision unless the precision value has been changed, in which case floating-point values print with the specified precision.

n) The ostream member function put outputs the specified number of characters.

o) The stream manipulators dec, oct and hex affect only the next integer output operation.

p) By default, memory addresses are displayed as long integers.

23.3 For each of the following, write a single statement that performs the indicated task.

a) Output the string "Enter your name: ".

b) Use a stream manipulator that causes the exponent in scientific notation and the letters in hexadecimal values to print in capital letters.

c) Output the address of the variable myString of type char *.

d) Use a stream manipulator to ensure that floating-point values print in scientific notation.

e) Output the address in variable integerPtr of type int *.

f) Use a stream manipulator such that, when integer values are output, the integer base for octal and hexadecimal values is displayed.

g) Output the value pointed to by floatPtr of type float *.

h) Use a stream member function to set the fill character to '*' for printing in field widths larger than the values being output. Repeat this statement with a stream manipulator.

i) Output the characters 'O' and 'K' in one statement with ostream function put.

j) Get the value of the next character to input without extracting it from the stream.

k) Input a single character into variable charValue of type char, using the istream member function get in two different ways.

l) Input and discard the next six characters in the input stream.

m) Use istream member function read to input 50 characters into char array line.

n) Read 10 characters into character array name. Stop reading characters if the '.' delimiter is encountered. Do not remove the delimiter from the input stream. Write another statement that performs this task and removes the delimiter from the input.
Use the `istream` member function `gcount` to determine the number of characters input into character array `line` by the last call to `istream` member function `read`, and output that number of characters, using `ostream` member function `write`.


Print the current precision setting, using a member function of object `cout`.

Input an integer value into `int` variable `months` and a floating-point value into `float` variable `percentageRate`.

Print the current precision setting, using a member function of object `cout`.

Print integer 100 in octal, hexadecimal and decimal, using stream manipulators and separated by tabs.

Print integer 100 in decimal, octal and hexadecimal separated by tabs, using a stream manipulator to change the base.

Read characters into character array `line` until the character 'z' is encountered, up to a limit of 20 characters (including a terminating null character). Do not extract the delimiter character from the stream.

Use integer variables `x` and `y` to specify the field width and precision used to display the double value 87.4573, and display the value.

Identify the error in each of the following statements and explain how to correct it.

- a) `cout << "Value of x <= y is: " << x <= y;`
- b) The following statement should print the integer value of 'c'.
  `cout << 'c';`
- c) `cout << ""A string in quotes"";`

For each of the following, show the output.

- a) `cout << "12345" << endl;`
- b) `cout.width( 5 );`
- c) `cout.fill( 'o' );`
- d) `cout << 123 << endl << 123;`
- e) `cout << setw( 10 ) << setfill( '$' ) << 10000;`
- f) `cout << setw( 8 ) << setprecision( 3 ) << 1024.987654;`
- g) `cout << showbase << oct << 99 << endl << hex << 99;`
- h) `cout << 100000 << endl << showpos << 100000;`
- i) `cout << setw( 10 ) << setprecision( 2 ) << scientific << 444.93738;`

**Answers to Self-Review Exercises**

23.1 a) streams. b) left, right and internal. c) flags. d) `<iostream>`. e) `<iomanip>`. f) `<fstream>`. g) write. h) `istream`. i) cerr or clog. j) `ofstream`. k) `<`. l) `cin`, `cout`, `cerr` and `clog`. m) `>>`. n) oct, hex and dec. o) `showpos`.

23.2 a) False. The stream member function `flags` with a `fmtflags` argument sets the `flags` state variable to its argument and returns the prior state settings. b) False. The stream insertion and stream extraction operators are not overloaded for all user-defined types. You must specifically provide the overloaded operator functions to overload the stream operators for use with each user-defined type you create. c) False. The stream member function `flags` with no arguments returns the current format settings as a `fmtflags` data type, which represents the format state. d) True. e) False. To overload the stream insertion operator `<<`, the overloaded operator function must take an `ostream` reference and a reference to a user-defined type as arguments and return an `ostream` reference. f) True. g) True. h) True. i) True. j) False. The `cin` stream is connected to the standard input.
of the computer, which normally is the keyboard. k) True. l) True. m) True. n) False. The `ostream` member function `put` outputs its single-character argument. o) False. The stream manipulators `dec`, `oct`, and `hex` set the output format state for integers to the specified base until the base is changed again or the program terminates. p) False. Memory addresses are displayed in hexadecimal format by default. To display addresses as `long` integers, the address must be cast to a `long` value.

23.3 a) `cout << "Enter your name: ";`
b) `cout << uppercase;`
c) `cout << static_cast<void*>(myString);`
d) `cout << scientific;`
e) `cout << integerPtr;`
f) `cout << showbase;`
g) `cout << *floatPtr;`
h) `cout.fill( 's' );`
i) `cout.put( 'O' ).put( 'K' );`
j) `cin.peek();`
k) `charValue = cin.get();`
l) `cin.get( charValue );`
m) `cin.ignore( 6 );`

23.4 a) Error: The precedence of the `<<` operator is higher than that of `<=`, which causes the statement to be evaluated improperly and also causes a compiler error. Correction: Place parentheses around the expression `x <= y`.
b) Error: In C++, characters are not treated as small integers, as they are in C. Correction: To print the numerical value for a character in the computer's character set, the character must be cast to an integer value, as in the following:

c) Error: Quote characters cannot be printed in a string unless an escape sequence is used. Correction: Print the string in one of the following ways:

23.5 a) 12345
   **123**
   123
b) $$$$$10000$
c) 1024.988
d) 0143
   0x63
Exercises

23.6 Write a statement for each of the following:
   a) Print integer 40000 left justified in a 15-digit field.
   b) Read a string into character array variable state.
   c) Print 200 with and without a sign.
   d) Print the decimal value 100 in hexadecimal form preceded by 0x.
   e) Read characters into array charArray until the character 'p' is encountered, up to a limit of 10 characters (including the terminating null character). Extract the delimiter from the input stream, and discard it.
   f) Print 1.234 in a 9-digit field with preceding zeros.

23.7 (Inputting Decimal, Octal and Hexadecimal Values) Write a program to test the inputting of integer values in decimal, octal and hexadecimal formats. Output each integer read by the program in all three formats. Test the program with the following input data: 10, 010, 0x10.

23.8 (Printing Pointer Values as Integers) Write a program that prints pointer values, using casts to all the integer data types. Which ones print strange values? Which ones cause errors?

23.9 (Printing with Field Widths) Write a program to test the results of printing the integer value 12345 and the floating-point value 1.2345 in various-sized fields. What happens when the values are printed in fields containing fewer digits than the values?

23.10 (Rounding) Write a program that prints the value 100.453627 rounded to the nearest digit, tenth, hundredth, thousandth and ten-thousandth.

23.11 Write a program that inputs a string from the keyboard and determines the length of the string. Print the string in a field width that is twice the length of the string.

23.12 (Converting Fahrenheit to Celsius) Write a program that converts integer Fahrenheit temperatures from 0 to 212 degrees to floating-point Celsius temperatures with 3 digits of precision. Use the formula

\[
celsius = \frac{5.0}{9.0} \times (\text{fahrenheit} - 32);
\]

to perform the calculation. The output should be printed in two right-justified columns and the Celsius temperatures should be preceded by a sign for both positive and negative values.

23.13 In some programming languages, strings are entered surrounded by either single or double quotation marks. Write a program that reads the three strings suzy, "suzy" and 'suzy'. Are the single and double quotes ignored or read as part of the string?

23.14 (Reading Phone Numbers with and Overloaded Stream Extraction Operator) In Fig. 19.5, the stream extraction and stream insertion operators were overloaded for input and output of objects of the PhoneNumber class. Rewrite the stream extraction operator to perform the following error checking on input. The operator>> function will need to be reimplemented.
   a) Input the entire phone number into an array. Test that the proper number of characters has been entered. There should be a total of 14 characters read for a phone number of the form (800) 555-1212. Use \texttt{ios\_base-member-function clear} to set failbit for improper input.
   b) The area code and exchange do not begin with 0 or 1. Test the first digit of the area-code and exchange portions of the phone number to be sure that neither begins with 0 or 1. Use \texttt{ios\_base-member-function clear} to set failbit for improper input.
c) The middle digit of an area code used to be limited to 0 or 1 (although this has changed recently). Test the middle digit for a value of 0 or 1. Use the `ios_base`-member-function `clear` to set `failbit` for improper input. If none of the above operations results in `failbit` being set for improper input, copy the three parts of the telephone number into the `areaCode`, `exchange` and `line` members of the `PhoneNumber` object. If `failbit` has been set on the input, have the program print an error message and end, rather than print the phone number.

23.15 (Point Class) Write a program that accomplishes each of the following:
   a) Create a user-defined class `Point` that contains the private integer data members `xCoordinate` and `yCoordinate` and declares stream insertion and stream extraction overloaded operator functions as `friends` of the class.
   b) Define the stream insertion and stream extraction operator functions. The stream extraction operator function should determine whether the data entered is valid, and, if not, it should set the `failbit` to indicate improper input. The stream insertion operator should not be able to display the point after an input error occurred.
   c) Write a `main` function that tests input and output of user-defined class `Point`, using the overloaded stream extraction and stream insertion operators.

23.16 (Complex Class) Write a program that accomplishes each of the following:
   a) Create a user-defined class `Complex` that contains the private integer data members `real` and `imaginary` and declares stream insertion and stream extraction overloaded operator functions as `friends` of the class.
   b) Define the stream insertion and stream extraction operator functions. The stream extraction operator function should determine whether the data entered is valid, and, if not, it should set `failbit` to indicate improper input. The input should be of the form $3 + 8i$
   c) The values can be negative or positive, and it’s possible that one of the two values is not provided, in which case the appropriate data member should be set to 0. The stream insertion operator should not be able to display the point if an input error occurred. For negative imaginary values, a minus sign should be printed rather than a plus sign.
   d) Write a `main` function that tests input and output of user-defined class `Complex`, using the overloaded stream extraction and stream insertion operators.

23.17 (Printing a Table of ASCII Values) Write a program that uses a `for` statement to print a table of ASCII values for the characters in the ASCII character set from 33 to 126. The program should print the decimal value, octal value, hexadecimal value and character value for each character. Use the stream manipulators `dec`, `oct` and `hex` to print the integer values.

23.18 Write a program to show that the `getline` and three-argument `get istream` member functions both end the input string with a string-terminating null character. Also, show that `get` leaves the delimiter character on the input stream, whereas `getline` extracts the delimiter character and discards it. What happens to the unread characters in the stream?
Exception Handling

It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.
—Franklin Delano Roosevelt

If they’re running and they don’t look where they’re going I have to come out from somewhere and catch them.
—Jerome David Salinger

I never forget a face, but in your case I’ll make an exception.
—Groucho Marx

Objectives
In this chapter you’ll learn:

■ What exceptions are and when to use them.
■ To use try, catch and throw to detect, handle and indicate exceptions, respectively.
■ To process uncaught and unexpected exceptions.
■ To declare new exception classes.
■ How stack unwinding enables exceptions not caught in one scope to be caught in another scope.
■ To handle new failures.
■ To use auto_ptr to prevent memory leaks.
■ To understand the standard exception hierarchy.
24.1 Introduction

In this chapter, we introduce exception handling. An exception is an indication of a problem that occurs during a program’s execution. The name “exception” implies that the problem occurs infrequently—if the “rule” is that a statement normally executes correctly, then the “exception to the rule” is that a problem occurs. Exception handling enables you to create applications that can resolve (or handle) exceptions. In many cases, handling an exception allows a program to continue executing as if no problem had been encountered. A more severe problem could prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem before terminating in a controlled manner. The features presented in this chapter enable you to write robust and fault-tolerant programs that can deal with problems that may arise and continue executing or terminate gracefully. The style and details of C++ exception handling are based in part on the work of Andrew Koenig and Bjarne Stroustrup, as presented in their paper, “Exception Handling for C++ (revised).”

Error-Prevention Tip 24.1
Exception handling helps improve a program’s fault tolerance.

Software Engineering Observation 24.1
Exception handling provides a standard mechanism for processing errors. This is especially important when working on a project with a large team of programmers.

We begin with an overview of exception-handling concepts, then demonstrates basic exception-handling techniques. We show these techniques via an example that demonstrates handling an exception that occurs when a function attempts to divide by zero. We then discuss additional exception-handling issues, such as how to handle exceptions that occur in a constructor or destructor and how to handle exceptions that occur if operator \texttt{new} fails to allocate memory for an object. We conclude the chapter by introducing several classes that the C++ Standard Library provides for handling exceptions.

24.2 Exception-Handling Overview

Program logic frequently tests conditions that determine how program execution proceeds. Consider the following pseudocode:

```
Perform a task
If the preceding task did not execute correctly
    Perform error processing
Perform next task
If the preceding task did not execute correctly
    Perform error processing
...
```

In this pseudocode, we begin by performing a task. We then test whether that task executed correctly. If not, we perform error processing. Otherwise, we continue with the next task. Although this form of error handling works, intermixing program logic with error-handling logic can make the program difficult to read, modify, maintain and debug—especially in large applications.

**Performance Tip 24.1**

If the potential problems occur infrequently, intermixing program logic and error-handling logic can degrade a program’s performance, because the program must (potentially frequently) perform tests to determine whether the task executed correctly and the next task can be performed.

Exception handling enables you to remove error-handling code from the “main line” of the program’s execution, which improves program clarity and enhances modifiability. You can decide to handle any exceptions you choose—all exceptions, all exceptions of a certain type or all exceptions of a group of related types (e.g., exception types that belong to an inheritance hierarchy). Such flexibility reduces the likelihood that errors will be overlooked and thereby makes a program more robust.

With programming languages that do not support exception handling, programmers often delay writing error-processing code or sometimes forget to include it. This results in less robust software products. C++ enables you to deal with exception handling easily from the inception of a project.

24.3 Example: Handling an Attempt to Divide by Zero

Let’s consider a simple example of exception handling (Figs. 24.1–24.2). The purpose of this example is to show how to prevent a common arithmetic problem—division by zero. In C++, division by zero using integer arithmetic typically causes a program to terminate prematurely. In floating-point arithmetic, some C++ implementations allow division by zero, in which case positive or negative infinity is displayed as INF or -INF, respectively.

In this example, we define a function named `quotient` that receives two integers input by the user and divides its first `int` parameter by its second `int` parameter. Before performing the division, the function casts the first `int` parameter’s value to type `double`. 
Then, the second int parameter’s value is promoted to type double for the calculation. So function quotient actually performs the division using two double values and returns a double result.

Although division by zero is allowed in floating-point arithmetic, for the purpose of this example we treat any attempt to divide by zero as an error. Thus, function quotient tests its second parameter to ensure that it isn’t zero before allowing the division to proceed. If the second parameter is zero, the function uses an exception to indicate to the caller that a problem occurred. The caller (main in this example) can then process the exception and allow the user to type two new values before calling function quotient again. In this way, the program can continue to execute even after an improper value is entered, thus making the program more robust.

The example consists of two files. DivideByZeroException.h (Fig. 24.1) defines an exception class that represents the type of the problem that might occur in the example, and fig24_02.cpp (Fig. 24.2) defines the quotient function and the main function that calls it. Function main contains the code that demonstrates exception handling.

**Defining an Exception Class to Represent the Type of Problem That Might Occur**

Figure 24.1 defines class DivideByZeroException as a derived class of Standard Library class runtime_error (defined in header file `<stdexcept>`). Class runtime_error—a derived class of Standard Library class exception (defined in header file `<exception>`)—is the C++ standard base class for representing runtime errors. Class exception is the standard C++ base class for all exceptions. (Section 24.13 discusses class exception and its derived classes in detail.) A typical exception class that derives from the runtime_error class defines only a constructor (e.g., lines 12–13) that passes an error-message string to the base-class runtime_error constructor. Every exception class that derives directly or indirectly from exception contains the virtual function what, which returns an exception object’s error message. You’re not required to derive a custom exception class, such as DivideByZeroException, from the standard exception classes provided by C++. However, doing so allows you to use the virtual function what to obtain an appropriate error message. We use an object of this DivideByZeroException class in Fig. 24.2 to indicate when an attempt is made to divide by zero.

```c++
// Fig. 24.1: DivideByZeroException.h
// Class DivideByZeroException definition.
#include <stdexcept> // stdexcept header file contains runtime_error
using namespace std;

// DivideByZeroException objects should be thrown by functions
// upon detecting division-by-zero exceptions
class DivideByZeroException : public runtime_error
{
  public:
    // constructor specifies default error message
    DivideByZeroException()
      : runtime_error("attempted to divide by zero") {}
}; // end class DivideByZeroException
```

**Fig. 24.1** | Class DivideByZeroException definition.
Demonstrating Exception Handling

The program in Fig. 24.2 uses exception handling to wrap code that might throw a "divide-by-zero" exception and to handle that exception, should one occur. The application enables the user to enter two integers, which are passed as arguments to function quotient (lines 10–18). This function divides its first parameter (numerator) by its second parameter (denominator). Assuming that the user does not specify 0 as the denominator for the division, function quotient returns the division result. However, if the user inputs 0 for the denominator, function quotient throws an exception. In the sample output, the first two lines show a successful calculation, and the next two lines show a failed calculation due to an attempt to divide by zero. When the exception occurs, the program informs the user of the mistake and prompts the user to input two new integers. After we discuss the code, we'll consider the user inputs and flow of program control that yield these outputs.

```cpp
// Fig. 24.2: Fig24_02.cpp
// A simple exception-handling example that checks for
// divide-by-zero exceptions.
#include <iostream>
#include "DivideByZeroException.h" // DivideByZeroException class
using namespace std;

// perform division and throw DivideByZeroException object if
divide-by-zero exception occurs
double quotient( int numerator, int denominator )
{
    // throw DivideByZeroException if trying to divide by zero
    if ( denominator == 0 )
        throw DivideByZeroException(); // terminate function

    // return division result
    return static_cast<double>( numerator ) / denominator;
}

int main()
{
    int number1; // user-specified numerator
    int number2; // user-specified denominator
    double result; // result of division

    cout << "Enter two integers (end-of-file to end): ";

    // enable user to enter two integers to divide
    while ( cin >> number1 >> number2 )
    {
        // try block contains code that might throw exception
        // and code that should not execute if an exception occurs
        try
        {
            result = quotient( number1, number2 );
        }
```

Fig. 24.2  Exception-handling example that throws exceptions on attempts to divide by zero. (Part 1 of 2.)
Chapter 24 Exception Handling

Enclosing Code in a try Block

The program begins by prompting the user to enter two integers. The integers are input in the condition of the while loop (line 29). Line 35 passes the values to function quotient (lines 10–18), which either divides the integers and returns a result, or throws an exception (i.e., indicates that an error occurred) on an attempt to divide by zero. Exception handling is geared to situations in which the function that detects an error is unable to handle it.

C++ provides try blocks to enable exception handling. A try block consists of keyword try followed by braces ({{}) that define a block of code in which exceptions might occur. The try block encloses statements that might cause exceptions and statements that should be skipped if an exception occurs.

A try block (lines 33–37) encloses the invocation of function quotient and the statement that displays the division result. In this example, because the invocation of function quotient (line 35) can throw an exception, we enclose this function invocation in a try block. Enclosing the output statement (line 36) in the try block ensures that the output will occur only if function quotient returns a result.

```cpp
36    cout << "The quotient is: " << result << endl;
37    } // end try
38    catch ( DivideByZeroException &divideByZeroException )
39    {
40        cout << "Exception occurred: "
41            << divideByZeroException.what() << endl;
42    } // end catch
43    cout << "Enter two integers (end-of-file to end): ";
44    } // end while
45    cout << endl;
46 } // end main
```

Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z

Fig. 24.2 | Exception-handling example that throws exceptions on attempts to divide by zero. (Part 2 of 2.)

Software Engineering Observation 24.2

Exceptions may surface through explicitly mentioned code in a try block, through calls to other functions and through deeply nested function calls initiated by code in a try block.

Defining a catch Handler to Process a DivideByZeroException

Exceptions are processed by catch handlers (also called exception handlers), which catch and handle exceptions. At least one catch handler (lines 38–42) must immediately follow
each try block. Each catch handler begins with the keyword catch and specifies in parentheses an exception parameter that represents the type of exception the catch handler can process (DivideByZeroException in this case). When an exception occurs in a try block, the catch handler that executes is the one whose type matches the type of the exception that occurred (i.e., the type in the catch block matches the thrown exception type exactly or is a base class of it). If an exception parameter includes an optional parameter name, the catch handler can use that parameter name to interact with the caught exception in the body of the catch handler, which is delimited by braces ({}). A catch handler typically reports the error to the user, logs it to a file, terminates the program gracefully or tries an alternate strategy to accomplish the failed task. In this example, the catch handler simply reports that the user attempted to divide by zero. Then the program prompts the user to enter two new integer values.

**Common Programming Error 24.1**

*It’s a syntax error to place code between a try block and its corresponding catch handlers or between its catch handlers.*

**Common Programming Error 24.2**

*Each catch handler can have only a single parameter—specifying a comma-separated list of exception parameters is a syntax error.*

**Common Programming Error 24.3**

*It’s a logic error to catch the same type in two different catch handlers following a single try block.*

**Termination Model of Exception Handling**

If an exception occurs as the result of a statement in a try block, the try block expires (i.e., terminates immediately). Next, the program searches for the first catch handler that can process the type of exception that occurred. The program locates the matching catch by comparing the thrown exception’s type to each catch’s exception-parameter type until the program finds a match. A match occurs if the types are identical or if the thrown exception’s type is a derived class of the exception-parameter type. When a match occurs, the code contained in the matching catch handler executes. When a catch handler finishes processing by reaching its closing right brace (}), the exception is considered handled and the local variables defined within the catch handler (including the catch parameter) go out of scope. Program control does not return to the point at which the exception occurred (known as the throw point), because the try block has expired. Rather, control resumes with the first statement (line 44) after the last catch handler following the try block. This is known as the termination model of exception handling. [Note: Some languages use the resumption model of exception handling, in which, after an exception is handled, control resumes just after the throw point.] As with any other block of code, when a try block terminates, local variables defined in the block go out of scope.

**Common Programming Error 24.4**

*Logic errors can occur if you assume that after an exception is handled, control will return to the first statement after the throw point.*
Chapter 24  Exception Handling

Error-Prevention Tip 24.2
With exception handling, a program can continue executing (rather than terminating) after dealing with a problem. This helps ensure the kind of robust applications that contribute to what's called mission-critical computing or business-critical computing.

If the try block completes its execution successfully (i.e., no exceptions occur in the try block), then the program ignores the catch handlers and program control continues with the first statement after the last catch following that try block.

If an exception that occurs in a try block has no matching catch handler, or if an exception occurs in a statement that is not in a try block, the function that contains the statement terminates immediately, and the program attempts to locate an enclosing try block in the calling function. This process is called stack unwinding and is discussed in Section 24.8.

Flow of Program Control When the User Enters a Nonzero Denominator
Consider the flow of control when the user inputs the numerator 100 and the denominator 7 (i.e., the first two lines of output in Fig. 24.2). In line 13, function quotient determines that the denominator does not equal zero, so line 17 performs the division and returns the result (14.2857) to line 35 as a double. Program control then continues sequentially from line 35, so line 36 displays the division result—line 37 ends the try block. Because the try block completed successfully and did not throw an exception, the program does not execute the statements contained in the catch handler (lines 38–42), and control continues to line 44 (the first line of code after the catch handler), which prompts the user to enter two more integers.

Flow of Program Control When the User Enters a Denominator of Zero
Now let's consider a more interesting case in which the user inputs the numerator 100 and the denominator 0 (i.e., the third and fourth lines of output in Fig. 24.2). In line 13, quotient determines that the denominator equals zero, which indicates an attempt to divide by zero. Line 14 throws an exception, which we represent as an object of class DivideByZeroException (Fig. 24.1).

To throw an exception, line 14 uses keyword throw followed by an operand that represents the type of exception to throw. Normally, a throw statement specifies one operand. (In Section 24.5, we discuss how to use a throw statement with no operand.) The operand of a throw can be of any type. If the operand is an object, we call it an exception object—in this example, the exception object is an object of type DivideByZeroException. However, a throw operand also can assume other values, such as the value of an expression that does not result in an object (e.g., throw x > 5) or the value of an int (e.g., throw 5). The examples in this chapter focus exclusively on throwing exception objects.

Common Programming Error 24.5
Use caution when throwing the result of a conditional expression (?:)—promotion rules could cause the value to be of a type different from the one expected. For example, when throwing an int or a double from the same conditional expression, the int is promoted to a double. So, a catch handler that catches an int would never execute based on such a conditional expression.
As part of throwing an exception, the throw operand is created and used to initialize the parameter in the catch handler, which we discuss momentarily. In this example, the throw statement in line 14 creates an object of class DivideByZeroException. When line 14 throws the exception, function quotient exits immediately. Therefore, line 14 throws the exception before function quotient can perform the division in line 17. This is a central characteristic of exception handling: A function should throw an exception before the error has an opportunity to occur.

Because we enclosed the call to quotient (line 35) in a try block, program control enters the catch handler (lines 38–42) that immediately follows the try block. This catch handler serves as the exception handler for the divide-by-zero exception. In general, when an exception is thrown within a try block, the exception is caught by a catch handler that specifies the type matching the thrown exception. In this program, the catch handler specifies that it catches DivideByZeroException objects—this type matches the object type thrown in function quotient. Actually, the catch handler catches a reference to the DivideByZeroException object created by function quotient’s throw statement (line 14). The exception object is maintained by the exception-handling mechanism.

Performance Tip 24.2
Catching an exception object by reference eliminates the overhead of copying the object that represents the thrown exception.

Good Programming Practice 24.1
Associating each type of runtime error with an appropriately named exception object improves program clarity.

The catch handler’s body (lines 40–41) prints the associated error message returned by calling function what of base-class runtime_error. This function returns the string that the DivideByZeroException constructor (lines 12–13 in Fig. 24.1) passed to the runtime_error base-class constructor.

24.4 When to Use Exception Handling

Exception handling is designed to process synchronous errors, which occur when a statement executes. Common examples of these errors are out-of-range array subscripts, arithmetic overflow (i.e., a value outside the representable range of values), division by zero, invalid function parameters and unsuccessful memory allocation (due to lack of memory). Exception handling is not designed to process errors associated with asynchronous events (e.g., disk I/O completions, network message arrivals, mouse clicks and keystrokes), which occur in parallel with, and independent of, the program’s flow of control.

Software Engineering Observation 24.3
Incorporate your exception-handling strategy into your system from inception. Including effective exception handling after a system has been implemented can be difficult.

Software Engineering Observation 24.4
Exception handling provides a single, uniform technique for processing problems. This helps programmers on large projects understand each other’s error-processing code.
Chapter 24  Exception Handling

The exception-handling mechanism also is useful for processing problems that occur when a program interacts with software elements, such as member functions, constructors, destructors and classes. Rather than handling problems internally, such software elements often use exceptions to notify programs when problems occur. This enables you to implement customized error handling for each application.

Software Engineering Observation 24.5
Avoid using exception handling as an alternate form of flow of control. These “additional” exceptions can “get in the way” of genuine error-type exceptions.

Software Engineering Observation 24.6
Exception handling enables predefined software components to communicate problems to application-specific components, which can then process the problems in an application-specific manner.

Performance Tip 24.3
When no exceptions occur, exception-handling code incurs little or no performance penalty. Thus, programs that implement exception handling operate more efficiently than do programs that intermix error-handling code with program logic.

Software Engineering Observation 24.7
Functions with common error conditions should return 0 or NULL (or other appropriate values) rather than throw exceptions. A program calling such a function can check the return value to determine success or failure of the function call.

Complex applications normally consist of predefined software components and application-specific components that use the predefined components. When a predefined component encounters a problem, that component needs a mechanism to communicate the problem to the application-specific component—the predefined component cannot know in advance how each application processes a problem that occurs.

24.5 Rethrowing an Exception
It’s possible that an exception handler, upon receiving an exception, might decide either that it cannot process that exception or that it can process the exception only partially. In such cases, the exception handler can defer the exception handling (or perhaps a portion of it) to another exception handler. In either case, you achieve this by rethrowing the exception via the statement

```
throw;
```

Regardless of whether a handler can process (even partially) an exception, the handler can rethrow the exception for further processing outside the handler. The next enclosing try block detects the rethrown exception, which a catch handler listed after that enclosing try block attempts to handle.

Common Programming Error 24.6
Executing an empty throw statement outside a catch handler calls function `terminate`, which abandons exception processing and terminates the program immediately.
Rethrowing an Exception

The program of Fig. 24.3 demonstrates rethrowing an exception. In main’s try block (lines 29–34), line 32 calls function throwException (lines 8–24). The throwException function also contains a try block (lines 11–15), from which the throw statement in line 14 throws an instance of standard-library-class exception. Function throwException’s catch handler (lines 16–21) catches this exception, prints an error message (lines 18–19) and rethrows the exception (line 20). This terminates function throwException and returns control to line 32 in the try...catch block in main. The try block terminates (so line 33 does not execute), and the catch handler in main (lines 35–38) catches this exception and prints an error message (line 37). [Note: Since we do not use the exception parameters in the catch handlers of this example, we omit the exception parameter names and specify only the type of exception to catch (lines 16 and 35).]

```c++
// Fig. 24.3: Fig24_03.cpp
// Demonstrating exception rethrowing.
#include <iostream>
#include <exception>
using namespace std;

// throw, catch and rethrow exception
void throwException()
{
  // throw exception and catch it immediately
  try
  {
    cout << "  Function throwException throws an exception\n";
    throw exception(); // generate exception
  } // end try
  catch ( exception & ) // handle exception
  {
    cout << "  Exception handled in function throwException\n";
    cout << "    Function throwException rethrows exception\n";
    throw; // rethrow exception for further processing
   } // end catch

  cout << "This also should not print\n";
} // end function throwException

int main()
{
  // throw exception
  try
  {
    cout << "main invokes function throwException\n";
    throwException();
    cout << "This should not print\n";
  } // end try
  catch ( exception & ) // handle exception
  {
    cout << "Exception handled in main\n";
  } // end catch

Fig. 24.3 | Rethrowing an exception. (Part 1 of 2.)
```
Chapter 24 Exception Handling

24.6 Exception Specifications

An optional exception specification (also called a throw list) enumerates a list of exceptions that a function can throw. For example, consider the function declaration

```cpp
int someFunction( double value )
    throw ( ExceptionA, ExceptionB, ExceptionC )
    {
        // function body
    }
```

In this definition, the exception specification, which begins with keyword throw immediately following the closing parenthesis of the function’s parameter list, indicates that function someFunction can throw exceptions of types ExceptionA, ExceptionB and ExceptionC. A function can throw only exceptions of the types indicated by the specification or exceptions of any type derived from these types. If the function throws an exception that does not belong to a specified type, the exception-handling mechanism calls function unexpected, which terminates the program.

A function that does not provide an exception specification can throw any exception. Placing throw()—an empty exception specification—after a function’s parameter list states that the function does not throw exceptions. If the function attempts to throw an exception, function unexpected is invoked. Section 24.7 shows how function unexpected can be customized by calling function set_unexpected. [Note: Some compilers ignore exception specifications.]

Common Programming Error 24.7
Throwing an exception that has not been declared in a function’s exception specification causes a call to function unexpected.

Error-Prevention Tip 24.3
The compiler will not generate a compilation error if a function contains a throw expression for an exception not listed in the function’s exception specification. An error occurs only when that function attempts to throw that exception at execution time. To avoid surprises at execution time, carefully check your code to ensure that functions do not throw exceptions not listed in their exception specifications.
24.7 Processing Unexpected Exceptions

Function unexpected calls the function registered with function set_unexpected (defined in header file `<exception>`). If no function has been registered in this manner, function terminate is called by default. Cases in which function terminate is called include:

1. The exception mechanism cannot find a matching catch for a thrown exception.
2. A destructor attempts to throw an exception during stack unwinding.
3. An attempt is made to rethrow an exception when there is no exception currently being handled.
4. A call to function unexpected defaults to calling function terminate. (Section 15.5.1 of the C++ Standard Document discusses several additional cases.) Function set_terminate can specify the function to invoke when terminate is called. Otherwise, terminate calls abort, which terminates the program without calling the destructors of any remaining objects of automatic or static storage class. This could lead to resource leaks when a program terminates prematurely.

Software Engineering Observation 24.8

It’s generally recommended that you do not use exception specifications unless you’re overriding a base-class member function that already has an exception specification. In this case, the exception specification is required for the derived-class member function.

Function set_terminate and function set_unexpected each return a pointer to the last function called by terminate and unexpected, respectively (0, the first time each is called). This enables you to save the function pointer so it can be restored later. Functions set_terminate and set_unexpected take as arguments pointers to functions with void return types and no arguments.

If the last action of a programmer-defined termination function is not to exit a program, function abort will be called to end program execution after the other statements of the programmer-defined termination function are executed.

Common Programming Error 24.8

Aborting a program component due to an uncaught exception could leave a resource—such as a file stream or an I/O device—in a state in which other programs are unable to acquire the resource. This is known as a resource leak.

24.8 Stack Unwinding

When an exception is thrown but not caught in a particular scope, the function call stack is “unwound,” and an attempt is made to catch the exception in the next outer try...catch block. Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function. If a try block encloses that statement, an attempt is made to catch the exception. If a try block does not enclose that statement, stack unwinding occurs again. If no catch handler ever catches this exception, function terminate is called to terminate the program. The program of Fig. 24.4 demonstrates stack unwinding.
// Fig. 24.4: Fig24_04.cpp
// Demonstrating stack unwinding.
#include <iostream>
#include <stdexcept>
using namespace std;

// function3 throws runtime error
void function3() throw ( runtime_error )
{
    cout << "In function 3" << endl;

    // no try block, stack unwinding occurs, return control to function2
    throw runtime_error( "runtime_error in function3" ); // no print
} // end function3

// function2 invokes function3
void function2() throw ( runtime_error )
{
    cout << "function3 is called inside function2" << endl;
    function3(); // stack unwinding occurs, return control to function1
} // end function2

// function1 invokes function2
void function1() throw ( runtime_error )
{
    cout << "function2 is called inside function1" << endl;
    function2(); // stack unwinding occurs, return control to main
} // end function1

// demonstrate stack unwinding
int main()
{
    // invoke function1
    try
    {
        cout << "function1 is called inside main" << endl;
        function1(); // call function1 which throws runtime_error
    } // end try
    catch ( runtime_error &error ) // handle runtime error
    {
        cout << "Exception occurred: " << error.what() << endl;
        cout << "Exception handled in main" << endl;
    } // end catch
} // end main

function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main

Fig. 24.4 | Stack unwinding.
In `main`, the try block (lines 34–38) calls `function1` (lines 24–28). Next, `function1` calls `function2` (lines 17–21), which in turn calls `function3` (lines 8–14). Line 13 of `function3` throws a `runtime_error` object. However, because no try block encloses the throw statement in line 13, stack unwinding occurs—`function3` terminates at line 13, then returns control to the statement in `function2` that invoked `function3` (i.e., line 20). Because no try block encloses line 20, stack unwinding occurs again—`function2` terminates at line 20 and returns control to the statement in `function1` that invoked `function2` (i.e., line 27). Because no try block encloses line 27, stack unwinding occurs one more time—`function1` terminates at line 27 and returns control to the statement in `main` that invoked `function1` (i.e., line 37). The try block of lines 34–38 encloses this statement, so the first matching catch handler located after this try block (line 39–43) catches and processes the exception. Line 41 uses `function what` to display the exception message. Recall that `function what` is a virtual function of class `exception` that can be overridden by a derived class to return an appropriate error message.

## 24.9 Constructors, Destructors and Exception Handling

First, let’s discuss an issue that we’ve mentioned but not yet resolved satisfactorily: What happens when an error is detected in a constructor? For example, how should an object’s constructor respond when `new` fails because it was unable to allocate required memory for storing that object’s internal representation? Because the constructor cannot return a value to indicate an error, we must choose an alternative means of indicating that the object has not been constructed properly. One scheme is to return the improperly constructed object and hope that anyone using it would make appropriate tests to determine that it’s in an inconsistent state. Another scheme is to set some variable outside the constructor. The preferred alternative is to require the constructor to throw an exception that contains the error information, thus offering an opportunity for the program to handle the failure.

Before an exception is thrown by a constructor, destructors are called for any member objects built as part of the object being constructed. Destructors are called for every automatic object constructed in a try block before an exception is thrown. Stack unwinding is guaranteed to have been completed at the point that an exception handler begins executing. If a destructor invoked as a result of stack unwinding throws an exception, terminate is called.

If an object has member objects, and if an exception is thrown before the outer object is fully constructed, then destructors will be executed for the member objects that have been constructed prior to the occurrence of the exception. If an array of objects has been partially constructed when an exception occurs, only the destructors for the constructed objects in the array will be called.

An exception could preclude the operation of code that would normally release a resource (such as memory or a file), thus causing a resource leak. One technique to resolve this problem is to initialize a local object to acquire the resource. When an exception occurs, the destructor for that object will be invoked and can free the resource.

### Error-Prevention Tip 24.4

*When an exception is thrown from the constructor for an object that is created in a new expression, the dynamically allocated memory for that object is released.*
24.10 Exceptions and Inheritance

Various exception classes can be derived from a common base class, as we discussed in Section 24.3, when we created class DivideByZeroException as a derived class of class exception. If a catch handler catches a pointer or reference to an exception object of a base-class type, it also can catch a pointer or reference to all objects of classes publicly derived from that base class—this allows for polymorphic processing of related errors.

Error-Prevention Tip 24.5

Using inheritance with exceptions enables an exception handler to catch related errors with concise notation. One approach is to catch each type of pointer or reference to a derived-class exception object individually, but a more concise approach is to catch pointers or references to base-class exception objects instead. Also, catching pointers or references to derived-class exception objects individually is error prone, especially if you forget to test explicitly for one or more of the derived-class pointer or reference types.

24.11 Processing new Failures

The C++ standard specifies that, when operator new fails, it throws a bad_alloc exception (defined in header file <new>). In this section, we present two examples of new failing. The first uses the version of new that throws a bad_alloc exception when new fails. The second uses function set_new_handler to handle new failures. [Note: The examples in Figs. 24.5–24.6 allocate large amounts of dynamic memory, which could cause your computer to become sluggish.]

new Throwing bad_alloc on Failure

Figure 24.5 demonstrates new throwing bad_alloc on failure to allocate the requested memory. The for statement (lines 16–20) inside the try block should loop 50 times and, on each pass, allocate an array of 50,000,000 double values. If new fails and throws a bad_alloc exception, the loop terminates, and the program continues in line 22, where the catch handler catches and processes the exception. Lines 24–25 print the message "Exception occurred:" followed by the message returned from the base-class-exception version of function what (i.e., an implementation-defined exception-specific message, such as "Allocation Failure" in Microsoft Visual C++). The output shows that the program performed only four iterations of the loop before new failed and threw the bad_alloc exception. Your output might differ based on the physical memory, disk space available for virtual memory on your system and the compiler you’re using.

```cpp
// Fig. 24.5: Fig24_05.cpp
// Demonstrating standard new throwing bad_alloc when memory
// cannot be allocated.
#include <iostream>
#include <new> // bad_alloc class is defined here
// bad_alloc class is defined here
using namespace std;
```

Fig. 24.5 | new throwing bad_alloc on failure. (Part 1 of 2.)
24.11 Processing new Failures

In old versions of C++, operator new returned 0 when it failed to allocate memory. The C++ standard specifies that standard-compliant compilers can continue to use a version of new that returns 0 upon failure. For this purpose, header file <new> defines object noexcept (of type noexcept_t), which is used as follows:

```cpp
double *ptr = new( noexcept ) double[ 50000000 ];
```

The preceding statement uses the version of new that does not throw bad_alloc exceptions (i.e., noexcept) to allocate an array of 50,000,000 doubles.

**Software Engineering Observation 24.9**
To make programs more robust, use the version of new that throws bad_alloc exceptions on failure.

**Handling new Failures Using Function set_new_handler**
An additional feature for handling new failures is function set_new_handler (prototyped in standard header file <new>). This function takes as its argument a pointer to a function that takes no arguments and returns void. This pointer points to the function that will be called if new fails. This provides you with a uniform approach to handling all new failures, regardless of where a failure occurs in the program. Once set_new_handler registers a new handler in the program, operator new does not throw bad_alloc on failure; rather, it defers the error handling to the new-handler function.

```cpp
int main()
{
    double *ptr[ 50 ];

    // aim each ptr[i] at a big block of memory
    try
    {
        // allocate memory for ptr[ i ]; new throws bad_alloc on failure
        for ( int i = 0; i < 50; i++ )
        {
            ptr[ i ] = new double[ 50000000 ]; // may throw exception
            cout << "ptr[" << i << "] points to 50,000,000 new doubles\n";
        } // end for
    } // end try
    catch ( bad_alloc &memoryAllocationException )
    {
        cerr << "Exception occurred: " << memoryAllocationException.what() << endl;
    } // end catch
} // end main
```

ptr[0] points to 50,000,000 new doubles
ptr[1] points to 50,000,000 new doubles
ptr[2] points to 50,000,000 new doubles
ptr[3] points to 50,000,000 new doubles
Exception occurred: bad allocation

Fig. 24.5 | new throwing bad_alloc on failure. (Part 2 of 2.)
If `new` allocates memory successfully, it returns a pointer to that memory. If `new` fails to allocate memory and `set_new_handler` did not register a new-handler function, `new` throws a `bad_alloc` exception. If `new` fails to allocate memory and a new-handler function has been registered, the new-handler function is called. The C++ standard specifies that the new-handler function should perform one of the following tasks:

1. Make more memory available by deleting other dynamically allocated memory (or telling the user to close other applications) and return to operator `new` to attempt to allocate memory again.

2. Throw an exception of type `bad_alloc`.

3. Call function `abort` or `exit` (both found in header file `<cstdlib>`) to terminate the program.

Figure 24.6 demonstrates `set_new_handler`. Function `customNewHandler` (lines 9–13) prints an error message (line 11), then calls `abort` (line 12) to terminate the program. The output shows that the loop iterated four times before `new` failed and invoked function `customNewHandler`. Your output might differ based on the physical memory, disk space available for virtual memory on your system and your compiler.

```cpp
// Fig. 24.6: Fig24_06.cpp
// Demonstrating set_new_handler.
#include <iostream>
#include <cstdlib> // abort function prototype
using namespace std;

void customNewHandler()
{
    cerr << "customNewHandler was called";
    abort();
} // end function customNewHandler

// using set_new_handler to handle failed memory allocation
int main()
{
    double *ptr[50];

    // specify that customNewHandler should be called on
    // memory allocation failure
    set_new_handler( customNewHandler );

    // aim each ptr[i] at a big block of memory; customNewHandler will be
    // called on failed memory allocation
    for ( int i = 0; i < 50; i++ )
    {
        ptr[ i ] = new double[50000000]; // may throw exception
        cout << "ptr[" << i << "] points to 50,000,000 new doubles\n";
    } // end for
} // end main
```

Fig. 24.6 | `set_new_handler` specifying the function to call when `new` fails. (Part 1 of 2.)
24.12 Class auto_ptr and Dynamic Memory Allocation

A common programming practice is to allocate dynamic memory, assign the address of that memory to a pointer, use the pointer to manipulate the memory and deallocate the memory with delete when the memory is no longer needed. If an exception occurs after successful memory allocation but before the delete statement executes, a memory leak could occur. The C++ standard provides class template auto_ptr in header file <memory> to deal with this situation.

An object of class auto_ptr maintains a pointer to dynamically allocated memory. When an auto_ptr object destructor is called (for example, when an auto_ptr object goes out of scope), it performs a delete operation on its pointer data member. Class template auto_ptr provides overloaded operators * and -> so that an auto_ptr object can be used just as a regular pointer variable is. Figure 24.9 demonstrates an auto_ptr object that points to a dynamically allocated object of class Integer (Figs. 24.7–24.8).

```cpp
// Fig. 24.7: Integer.h
// Integer class definition.

class Integer
{
  public:
    Integer( int i = 0 ); // Integer default constructor
    ~Integer(); // Integer destructor
    void setInteger( int i ); // functions to set Integer
    int getInteger() const; // function to return Integer
  private:
    int value;
}; // end class Integer

// Fig. 24.8: Integer.cpp
// Integer member function definitions.

#include <iostream>
#include "Integer.h"
using namespace std;

Fig. 24.7 | Integer class definition.

Fig. 24.8 | Member function definitions of class Integer. (Part 1 of 2.)
```
Line 15 of Fig. 24.9 creates auto_ptr object ptrToInteger and initializes it with a pointer to a dynamically allocated Integer object that contains the value 7. Line 18 uses the auto_ptr overloaded -> operator to invoke function setInteger on the Integer object that ptrToInteger manages. Line 21 uses the auto_ptr overloaded * operator to dereference ptrToInteger, then uses the dot (.) operator to invoke function getInteger on the Integer object. Like a regular pointer, an auto_ptr’s -> and * overloaded operators can be used to access the object to which the auto_ptr points.
Because ptrToInteger is a local automatic variable in main, ptrToInteger is destroyed when main terminates. The auto_ptr destructor forces a delete of the Integer object pointed to by ptrToInteger, which in turn calls the Integer class destructor. The memory that Integer occupies is released, regardless of how control leaves the block (e.g., by a return statement or by an exception). Most importantly, using this technique can prevent memory leaks. For example, suppose a function returns a pointer aimed at some object. Unfortunately, the function caller that receives this pointer might not delete the object, thus resulting in a memory leak. However, if the function returns an auto_ptr to the object, the object will be deleted automatically when the auto_ptr object’s destructor gets called.

Only one auto_ptr at a time can own a dynamically allocated object and the object cannot be an array. By using its overloaded assignment operator or copy constructor, an auto_ptr can transfer ownership of the dynamic memory it manages. The last auto_ptr object that maintains the pointer to the dynamic memory will delete the memory. This makes auto_ptr an ideal mechanism for returning dynamically allocated memory to client code. When the auto_ptr goes out of scope in the client code, the auto_ptr’s destructor deletes the dynamic memory.

24.13 Standard Library Exception Hierarchy

Experience has shown that exceptions fall nicely into a number of categories. The C++ Standard Library includes a hierarchy of exception classes, some of which are shown in Fig. 24.10. As we first discussed in Section 24.3, this hierarchy is headed by base-class ex-

```cpp
16  cout << "\nUsing the auto_ptr to manipulate the Integer\n";
17  ptrToInteger->setInteger( 99 ); // use auto_ptr to set Integer value
18  // use auto_ptr to get Integer value
19  cout << "Integer after setInteger: " << ( *ptrToInteger ).getInteger()
20 } // end main
```

Fig. 24.9 | auto_ptr object manages dynamically allocated memory. (Part 2 of 2.)

Because ptrToInteger is a local automatic variable in main, ptrToInteger is destroyed when main terminates. The auto_ptr destructor forces a delete of the Integer object pointed to by ptrToInteger, which in turn calls the Integer class destructor. The memory that Integer occupies is released, regardless of how control leaves the block (e.g., by a return statement or by an exception). Most importantly, using this technique can prevent memory leaks. For example, suppose a function returns a pointer aimed at some object. Unfortunately, the function caller that receives this pointer might not delete the object, thus resulting in a memory leak. However, if the function returns an auto_ptr to the object, the object will be deleted automatically when the auto_ptr object’s destructor gets called.

Only one auto_ptr at a time can own a dynamically allocated object and the object cannot be an array. By using its overloaded assignment operator or copy constructor, an auto_ptr can transfer ownership of the dynamic memory it manages. The last auto_ptr object that maintains the pointer to the dynamic memory will delete the memory. This makes auto_ptr an ideal mechanism for returning dynamically allocated memory to client code. When the auto_ptr goes out of scope in the client code, the auto_ptr’s destructor deletes the dynamic memory.

24.13 Standard Library Exception Hierarchy

Experience has shown that exceptions fall nicely into a number of categories. The C++ Standard Library includes a hierarchy of exception classes, some of which are shown in Fig. 24.10. As we first discussed in Section 24.3, this hierarchy is headed by base-class ex-
ception (defined in header file `<exception>`), which contains virtual function `what`, which derived classes can override to issue appropriate error messages.

Immediate derived classes of base-class exception include `runtime_error` and `logic_error` (both defined in header `<stdexcept>`), each of which has several derived classes. Also derived from exception are the exceptions thrown by C++ operators—for example, `bad_alloc` is thrown by `new` (Section 24.11), `bad_cast` is thrown by `dynamic_cast` (Chapter 21) and `bad_typeid` is thrown by `typeid` (Chapter 21). Including `bad_exception` in the throw list of a function means that, if an unexpected exception occurs, function `unexpected` can throw `bad_exception` rather than terminating the program’s execution (by default) or calling another function specified by `set_unexpected`.

Class `logic_error` is the base class of several standard exception classes that indicate errors in program logic. For example, class `invalid_argument` indicates that an invalid argument was passed to a function. (Proper coding can, of course, prevent invalid arguments from reaching a function.) Class `length_error` indicates that a length larger than the maximum size allowed for the object being manipulated was used for that object. Class `out_of_range` indicates that a value, such as a subscript into an array, exceeded its allowed range of values.

Class `runtime_error`, which we used briefly in Section 24.8, is the base class of several other standard exception classes that indicate execution-time errors. For example, class `overflow_error` describes an arithmetic overflow error (i.e., the result of an arithmetic operation is larger than the largest number that can be stored in the computer) and class

![Diagram of the Standard Library exception classes](image-url)
underflow_error describes an arithmetic underflow error (i.e., the result of an arithmetic operation is smaller than the smallest number that can be stored in the computer).

**Common Programming Error 24.11**
Exception classes need not be derived from class exception, so catching type exception is not guaranteed to catch all exceptions a program could encounter.

**Error-Prevention Tip 24.6**
To catch all exceptions potentially thrown in a try block, use catch(...). One weakness with catching exceptions in this way is that the type of the caught exception is unknown at compile time. Another weakness is that, without a named parameter, there is no way to refer to the exception object inside the exception handler.

**Software Engineering Observation 24.10**
The standard exception hierarchy is a good starting point for creating exceptions. You can build programs that can throw standard exceptions, throw exceptions derived from the standard exceptions or throw your own exceptions not derived from the standard exceptions.

**Software Engineering Observation 24.11**
Use catch(...) to perform recovery that does not depend on the exception type (e.g., releasing common resources). The exception can be rethrown to alert more specific enclosing catch handlers.

### 24.14 Other Error-Handling Techniques

We’ve discussed several ways to deal with exceptional situations prior to this chapter. The following summarizes these and other error-handling techniques:

- **Ignore the exception.** If an exception occurs, the program might fail as a result of the uncaught exception. This is devastating for commercial software products and special-purpose mission-critical software, but, for software developed for your own purposes, ignoring many kinds of errors is common.

- **Abort the program.** This, of course, prevents a program from running to completion and producing incorrect results. For many types of errors, this is appropriate, especially for nonfatal errors that enable a program to run to completion (potentially misleading you to think that the program functioned correctly). This strategy is inappropriate for mission-critical applications. Resource issues also are important here—if a program obtains a resource, the program should release that resource before program termination.

- **Set error indicators.** The problem with this approach is that programs might not check these error indicators at all points at which the errors could be troublesome. Another problem is that the program, after processing the problem, might not clear the error indicators.

- **Test for the error condition, issue an error message and call exit (in <cstdlib>) to pass an appropriate error code to the program’s environment.**
• Certain kinds of errors have dedicated capabilities for handling them. For example, when operator `new` fails to allocate memory, a `new_handler` function can be called to handle the error. This function can be customized by supplying a function name as the argument to `set_new_handler`, as we discussed in Section 24.11.

### 24.15 Wrap-Up

In this chapter, you learned how to use exception handling to deal with errors in a program. You learned that exception handling enables you to remove error-handling code from the “main line” of the program’s execution. We demonstrated exception handling in the context of a divide-by-zero example. We also showed how to use `try` blocks to enclose code that may throw an exception, and how to use `catch` handlers to deal with exceptions that may arise. You learned how to throw and rethrow exceptions, and how to handle the exceptions that occur in constructors. The chapter continued with discussions of processing `new` failures, dynamic memory allocation with class `auto_ptr` and the standard library exception hierarchy.

### Summary

**Section 24.1 Introduction**
- An exception is an indication of a problem that occurs during a program’s execution.
- Exception handling enables you to create programs that can resolve problems that occur at execution time—often allowing programs to continue executing as if no problems had been encountered. More severe problems may require a program to notify the user of the problem before terminating in a controlled manner.

**Section 24.2 Exception-Handling Overview**
- Exception handling enables you to remove error-handling code from the “main line” of the program’s execution, which improves program clarity and enhances modifiability.

**Section 24.3 Example: Handling an Attempt to Divide by Zero**
- Class `exception` is the standard C++ base class for exceptions. Class `exception` provides virtual function `what` that returns an appropriate error message and can be overridden in derived classes.
- Class `runtime_error` (defined in header `<stdexcept>`) is the C++ standard base class for representing runtime errors.
- C++ uses the termination model of exception handling.
- A `try` block consists of keyword `try` followed by braces `{}` that define a block of code in which exceptions might occur. The `try` block encloses statements that might cause exceptions and statements that should not execute if exceptions occur.
- At least one `catch` handler must immediately follow a `try` block. Each `catch` handler specifies an exception parameter that represents the type of exception the `catch` handler can process.
- If an exception parameter includes an optional parameter name, the `catch` handler can use that parameter name to interact with a caught exception object.
- The point in the program at which an exception occurs is called the throw point.
• If an exception occurs in a try block, the try block expires and program control transfers to the first catch in which the exception parameter’s type matches that of the thrown exception.

• When a try block terminates, local variables defined in the block go out of scope.

• When a try block terminates due to an exception, the program searches for the first catch handler that matches the type of exception that occurred. A match occurs if the types are identical or if the thrown exception’s type is a derived class of the exception-parameter type. When a match occurs, the code contained within the matching catch handler executes.

• When a catch handler finishes processing, the catch parameter and local variables defined within the catch handler go out of scope. Any remaining catch handlers that correspond to the try block are ignored, and execution resumes at the first line of code after the try...catch sequence.

• If no exceptions occur in a try block, the program ignores the catch handler(s) for that block. Program execution resumes with the next statement after the try...catch sequence.

• If an exception that occurs in a try block has no matching catch handler, or if an exception occurs in a statement that is not in a try block, the function that contains the statement terminates immediately, and the program attempts to locate an enclosing try block in the calling function. This process is called stack unwinding.

• To throw an exception, use keyword throw followed by an operand that represents the type of exception to throw. The operand of a throw can be of any type.

Section 24.4 When to Use Exception Handling

• Exception handling is for synchronous errors, which occur when a statement executes.

• Exception handling is not designed to process errors associated with asynchronous events, which occur in parallel with, and independent of, the program’s flow of control.

Section 24.5 Rethrowing an Exception

• The exception handler can defer the exception handling (or perhaps a portion of it) to another exception handler. In either case, the handler achieves this by rethrowing the exception.

• Common examples of exceptions are out-of-range array subscripts, arithmetic overflow, division by zero, invalid function parameters and unsuccessful memory allocations.

Section 24.6 Exception Specifications

• An optional exception specification enumerates a list of exceptions that a function can throw. A function can throw only exceptions of the types indicated by the exception specification or exceptions of any type derived from these types. If the function throws an exception that does not belong to a specified type, function unexpected is called and the program terminates.

• A function with no exception specification can throw any exception. The empty exception specification throw() indicates that a function does not throw exceptions. If a function with an empty exception specification attempts to throw an exception, function unexpected is invoked.

Section 24.7 Processing Unexpected Exceptions

• Function unexpected calls the function registered with function set_unexpected. If no function has been registered in this manner, function terminate is called by default.

• Function set_terminate can specify the function to invoke when terminate is called. Otherwise, terminate calls abort, which terminates the program without calling the destructors of objects that are declared static and auto.

• Functions set_terminate and set_unexpected each return a pointer to the last function called by terminate and unexpected, respectively (0, the first time each is called). This enables you to save the function pointer so it can be restored later.
Functions `set_terminate` and `set_unexpected` take as arguments pointers to functions with `void` return types and no arguments.

If a programmer-defined termination function does not exit a program, function `abort` will be called after the programmer-defined termination function completes execution.

Section 24.8 Stack Unwinding

Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function.

Section 24.9 Constructors, Destructors and Exception Handling

Exceptions thrown by a constructor cause destructors to be called for any objects built as part of the object being constructed before the exception is thrown.

Each automatic object constructed in a `try` block is destructed before an exception is thrown.

Stack unwinding completes before an exception handler begins executing.

If a destructor invoked as a result of stack unwinding throws an exception, `terminate` is called.

If an object has member objects, and if an exception is thrown before the outer object is fully constructed, then destructors will be executed for the member objects that have been constructed before the exception occurs.

If an array of objects has been partially constructed when an exception occurs, only the destructors for the constructed array element objects will be called.

When an exception is thrown from the constructor for an object that is created in a `new` expression, the dynamically allocated memory for that object is released.

Section 24.10 Exceptions and Inheritance

If a `catch` handler catches a pointer or reference to an exception object of a base-class type, it also can catch a pointer or reference to all objects of classes derived publicly from that base class—this allows for polymorphic processing of related errors.

Section 24.11 Processing `new` Failures

The C++ standard document specifies that, when operator `new` fails, it throws a `bad_alloc` exception (defined in header file `<new>`).

Function `set_new_handler` takes as its argument a pointer to a function that takes no arguments and returns `void`. This pointer points to the function that will be called if `new` fails.

Once `set_new_handler` registers a new handler in the program, operator `new` does not throw `bad_alloc` on failure; rather, it defers the error handling to the `new-handler` function.

If `new` allocates memory successfully, it returns a pointer to that memory.

If an exception occurs after successful memory allocation but before the `delete` statement executes, a memory leak could occur.

Section 24.12 Class `auto_ptr` and Dynamic Memory Allocation

The C++ Standard Library provides class template `auto_ptr` to deal with memory leaks.

An object of class `auto_ptr` maintains a pointer to dynamically allocated memory. An `auto_ptr` object’s destructor performs a `delete` operation on the `auto_ptr`’s pointer data member.

Class template `auto_ptr` provides overloaded operators `*` and `->` so that an `auto_ptr` object can be used just as a regular pointer variable is. An `auto_ptr` also transfers ownership of the dynamic memory it manages via its copy constructor and overloaded assignment operator.
**Section 24.13 Standard Library Exception Hierarchy**

- The C++ Standard Library includes a hierarchy of exception classes. This hierarchy is headed by base-class `exception`.
- Immediate derived classes of base class `exception` include `runtime_error` and `logic_error` (both defined in header `<stdexcept>`), each of which has several derived classes.
- Several operators throw standard exceptions—operator `new` throws `bad_alloc`, operator `dynamic_cast` throws `bad_cast` and operator `typeid` throws `bad_typeid`.
- Including `bad_exception` in the throw list of a function means that, if an unexpected exception occurs, function `unexpected` can throw `bad_exception` rather than terminating the program’s execution or calling another function specified by `set_unexpected`.

**Terminology**

- `abort` function 901
- arithmetic overflow error 910
- arithmetic underflow error 911
- asynchronous event 897
- `auto_ptr` class template 907
- `bad_alloc` exception 904
- `bad_cast` exception 910
- `bad_exception` exception 910
- `bad_typeid` exception 910
- catch handler 894
- catch keyword 895
- empty exception specification 900
- exception 890
- exception class 892
- exception handler 894
- exception handling 890
- `<exception>` header file 892
- exception object 896
- exception parameter 895
- exception specification 900
- fault-tolerant program 890
- `invalid_argument` exception 910
- `length_error` exception 910
- `logic_error` exception 910
- `<memory>` header file 907
- `new` handler 905
- `nothrow` object 905
- `out_of_range` exception 910
- `overflow_error` exception 910
- resource leak 901
- resumption model of exception handling 895
- rethrowing the exception 898
- robust program 890
- `runtime_error` exception 892
- `set_new_handler` function 904
- `set_terminate` function 901
- `set_unexpected` function 900
- stack unwinding 896
- `<stdexcept>` header file 892
- synchronous error 897
- `terminate` function 898
- termination model of exception handling 895
- throw 896
- throws an exception 894
- throw keyword 896
- throw list 900
- throw point 895
- `try` block 894
- `try` keyword 894
- `underflow_error` exception 911
- `unexpected` function 900
- what virtual function of class `exception` 892

**Self-Review Exercises**

**24.1** List five common examples of exceptions.

**24.2** Give several reasons why exception-handling techniques should not be used for conventional program control.

**24.3** Why are exceptions appropriate for dealing with errors produced by library functions?

**24.4** What’s a “resource leak”?

**24.5** If no exceptions are thrown in a try block, where does control proceed to after the try block completes execution?
Chapter 24 Exception Handling

24.6 What happens if an exception is thrown outside a try block?
24.7 Give a key advantage and a key disadvantage of using catch(…).
24.8 What happens if no catch handler matches the type of a thrown object?
24.9 What happens if several handlers match the type of the thrown object?
24.10 Why would you specify a base-class type as the type of a catch handler, then throw objects of derived-class types?
24.11 Suppose a catch handler with a precise match to an exception object type is available. Under what circumstances might a different handler be executed for exception objects of that type?
24.12 Must throwing an exception cause program termination?
24.13 What happens when a catch handler throws an exception?
24.14 What does the statement throw; do?
24.15 How do you restrict the exception types that a function can throw?
24.16 What happens if a function throws an exception of a type not allowed by the exception specification for the function?
24.17 What happens to the automatic objects that have been constructed in a try block when that block throws an exception?

Answers to Self-Review Exercises

24.1 Insufficient memory to satisfy a new request, array subscript out of bounds, arithmetic overflow, division by zero, invalid function parameters.
24.2 (a) Exception handling is designed to handle infrequently occurring situations that often result in program termination, so compiler writers are not required to implement exception handling to perform optimally. (b) Flow of control with conventional control structures generally is clearer and more efficient than with exceptions. (c) Problems can occur because the stack is unwound when an exception occurs and resources allocated prior to the exception might not be freed. (d) The “additional” exceptions make it more difficult for you to handle the larger number of exception cases.
24.3 It’s unlikely that a library function will perform error processing that will meet the unique needs of all users.
24.4 A program that terminates abruptly could leave a resource in a state in which other programs would not be able to acquire the resource, or the program itself might not be able to reacquire a “leaked” resource.
24.5 The exception handlers (in the catch handlers) for that try block are skipped, and the program resumes execution after the last catch handler.
24.6 An exception thrown outside a try block causes a call to terminate.
24.7 The form catch(…) catches any type of exception thrown in a try block. An advantage is that all possible exceptions will be caught. A disadvantage is that the catch has no parameter, so it cannot reference information in the thrown object and cannot know the cause of the exception.
24.8 This causes the search for a match to continue in the next enclosing try block if there is one. As this process continues, it might eventually be determined that there is no handler in the program that matches the type of the thrown object; in this case, terminate is called, which by default calls abort. An alternative terminate function can be provided as an argument to set_terminate.
24.9 The first matching exception handler after the try block is executed.
Exercises

24.10 This is a nice way to catch related types of exceptions.

24.11 A base-class handler would catch objects of all derived-class types.

24.12 No, but it does terminate the block in which the exception is thrown.

24.13 The exception will be processed by a catch handler (if one exists) associated with the try block (if one exists) enclosing the catch handler that caused the exception.

24.14 It rethrows the exception if it appears in a catch handler; otherwise, function unexpected is called.

24.15 Provide an exception specification listing the exception types that the function can throw.

24.16 Function unexpected is called.

24.17 The try block expires, causing destructors to be called for each of these objects.

Exercises

24.18 List various exceptional conditions that have occurred throughout this text. List as many additional exceptional conditions as you can. For each of these exceptions, describe briefly how a program typically would handle the exception, using the exception-handling techniques discussed in this chapter. Some typical exceptions are division by zero, arithmetic overflow, array subscript out of bounds, exhaustion of the free store, etc.

24.19 Under what circumstances would you not provide a parameter name when defining the type of the object that will be caught by a handler?

24.20 A program contains the statement

```cpp
throw;
```

Where would you normally expect to find such a statement? What if that statement appeared in a different part of the program?

24.21 Compare and contrast exception handling with the various other error-processing schemes discussed in the text.

24.22 Why should exceptions not be used as an alternate form of program control?

24.23 Describe a technique for handling related exceptions.

24.24 (Throwing Exceptions from a catch) Suppose a program throws an exception and the appropriate exception handler begins executing. Now suppose that the exception handler itself throws the same exception. Does this create infinite recursion? Write a program to check your observation.

24.25 (Catching Derived-Class Exceptions) Use inheritance to create various derived classes of runtime_error. Then show that a catch handler specifying the base class can catch derived-class exceptions.

24.26 (Throwing the Result of a Conditional Expression) Throw the result of a conditional expression that returns either a double or an int. Provide an int catch handler and a double catch handler. Show that only the double catch handler executes, regardless of whether the int or the double is returned.

24.27 (Local Variable Destructors) Write a program illustrating that all destructors for objects constructed in a block are called before an exception is thrown from that block.

24.28 (Member Object Destructors) Write a program illustrating that member object destructors are called for only those member objects that were constructed before an exception occurred.

24.29 (Catching All Exceptions) Write a program that demonstrates several exception types being caught with the catch(...) exception handler.
24.30 (Order of Exception Handlers) Write a program illustrating that the order of exception handlers is important. The first matching handler is the one that executes. Attempt to compile and run your program two different ways to show that two different handlers execute with two different effects.

24.31 (Constructors Throwing Exceptions) Write a program that shows a constructor passing information about constructor failure to an exception handler after a try block.

24.32 (Rethrowing Exceptions) Write a program that illustrates rethrowing an exception.

24.33 (Uncaught Exceptions) Write a program that illustrates that a function with its own try block does not have to catch every possible error generated within the try. Some exceptions can slip through to, and be handled in, outer scopes.

24.34 (Stack Unwinding) Write a program that throws an exception from a deeply nested function and still has the catch handler following the try block enclosing the call chain catch the exception.
Operators are shown in decreasing order of precedence from top to bottom (Figs. A.1–A.2).

<table>
<thead>
<tr>
<th>C Operator</th>
<th>Type</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>parentheses (function call operator)</td>
<td>left to right</td>
</tr>
<tr>
<td>[]</td>
<td>array subscript</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>member selection via object</td>
<td></td>
</tr>
<tr>
<td>-&gt;</td>
<td>member selection via pointer</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>unary postincrement</td>
<td></td>
</tr>
<tr>
<td>--</td>
<td>unary postdecrement</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>unary preincrement</td>
<td>right to left</td>
</tr>
<tr>
<td>--</td>
<td>unary predecrement</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>unary plus</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>unary minus</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>unary logical negation</td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>unary bitwise complement</td>
<td></td>
</tr>
<tr>
<td>( type )</td>
<td>C-style unary cast</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>dereference</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>sizeof</td>
<td>determine size in bytes</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>left to right</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>left to right</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>bitwise left shift</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>bitwise right shift</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>relational less than</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;=</td>
<td>relational less than or equal to</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>relational greater than</td>
<td></td>
</tr>
<tr>
<td>=&gt;</td>
<td>relational greater than or equal to</td>
<td></td>
</tr>
</tbody>
</table>

Fig. A.1  C operator precedence chart. (Part 1 of 2.)
### Appendix A Operator Precedence Charts

<table>
<thead>
<tr>
<th>C Operator</th>
<th>Type</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>relational is equal to</td>
<td>left to right</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>relational is not equal to</td>
<td>left to right</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>bitwise AND</td>
<td>left to right</td>
</tr>
<tr>
<td><code>^</code></td>
<td>bitwise exclusive OR</td>
<td>left to right</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>bitwise inclusive OR</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>logical AND</td>
<td>left to right</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td><code>?:</code></td>
<td>ternary conditional</td>
<td>right to left</td>
</tr>
<tr>
<td><code>=</code></td>
<td>assignment</td>
<td>right to left</td>
</tr>
<tr>
<td><code>+=</code></td>
<td>addition assignment</td>
<td>right to left</td>
</tr>
<tr>
<td><code>-=</code></td>
<td>subtraction assignment</td>
<td>right to left</td>
</tr>
<tr>
<td><code>*=</code></td>
<td>multiplication assignment</td>
<td>right to left</td>
</tr>
<tr>
<td><code>/=</code></td>
<td>division assignment</td>
<td>right to left</td>
</tr>
<tr>
<td><code>%=</code></td>
<td>modulus assignment</td>
<td>right to left</td>
</tr>
<tr>
<td><code>&amp;=</code></td>
<td>bitwise AND assignment</td>
<td>right to left</td>
</tr>
<tr>
<td><code>^=</code></td>
<td>bitwise exclusive OR assignment</td>
<td>right to left</td>
</tr>
<tr>
<td>`</td>
<td>=`</td>
<td>bitwise inclusive OR assignment</td>
</tr>
<tr>
<td><code>&lt;&lt;=</code></td>
<td>bitwise left shift assignment</td>
<td>right to left</td>
</tr>
<tr>
<td><code>&gt;&gt;=</code></td>
<td>bitwise right shift with sign</td>
<td>right to left</td>
</tr>
<tr>
<td><code>,</code></td>
<td>comma</td>
<td>left to right</td>
</tr>
</tbody>
</table>

**Fig. A.1** | C operator precedence chart. (Part 2 of 2.)

<table>
<thead>
<tr>
<th>C++ Operator</th>
<th>Type</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>::</code></td>
<td>binary scope resolution</td>
<td>left to right</td>
</tr>
<tr>
<td><code>::</code></td>
<td>unary scope resolution</td>
<td>left to right</td>
</tr>
<tr>
<td><code>()</code></td>
<td>parentheses (function call operator)</td>
<td>left to right</td>
</tr>
<tr>
<td><code>[]</code></td>
<td>array subscript</td>
<td>left to right</td>
</tr>
<tr>
<td><code>.</code></td>
<td>member selection via object</td>
<td>left to right</td>
</tr>
<tr>
<td><code>-&gt;</code></td>
<td>member selection via pointer</td>
<td>left to right</td>
</tr>
<tr>
<td><code>++</code></td>
<td>unary postincrement</td>
<td>left to right</td>
</tr>
<tr>
<td><code>--</code></td>
<td>unary postdecrement</td>
<td>left to right</td>
</tr>
<tr>
<td><code>typeid</code></td>
<td>runtime type information</td>
<td>left to right</td>
</tr>
<tr>
<td><code>dynamic_cast&lt;type&gt;</code></td>
<td>runtime type-checked cast</td>
<td>left to right</td>
</tr>
<tr>
<td><code>static_cast&lt;type&gt;</code></td>
<td>compile-time type-checked cast</td>
<td>left to right</td>
</tr>
<tr>
<td><code>reinterpret_cast&lt;type&gt;</code></td>
<td>cast for nonstandard conversions</td>
<td>left to right</td>
</tr>
<tr>
<td><code>const_cast&lt;type&gt;</code></td>
<td>cast away const-ness</td>
<td>left to right</td>
</tr>
</tbody>
</table>

**Fig. A.2** | C++ operator precedence chart. (Part 1 of 3.)
<table>
<thead>
<tr>
<th>C++ Operator</th>
<th>Type</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>unary preincrement</td>
<td>right to left</td>
</tr>
<tr>
<td>--</td>
<td>unary predecrement</td>
<td>right to left</td>
</tr>
<tr>
<td>+</td>
<td>unary plus</td>
<td>left to right</td>
</tr>
<tr>
<td>-</td>
<td>unary minus</td>
<td>left to right</td>
</tr>
<tr>
<td>!</td>
<td>unary logical negation</td>
<td>left to right</td>
</tr>
<tr>
<td>~</td>
<td>unary bitwise complement</td>
<td>left to right</td>
</tr>
<tr>
<td>(type)</td>
<td>C-style unary cast</td>
<td>left to right</td>
</tr>
<tr>
<td>sizeof</td>
<td>determine size in bytes</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>address</td>
<td>left to right</td>
</tr>
<tr>
<td>*</td>
<td>dereference</td>
<td>left to right</td>
</tr>
<tr>
<td>new</td>
<td>dynamic memory allocation</td>
<td>left to right</td>
</tr>
<tr>
<td>new[]</td>
<td>dynamic array allocation</td>
<td>left to right</td>
</tr>
<tr>
<td>delete</td>
<td>dynamic memory deallocation</td>
<td>left to right</td>
</tr>
<tr>
<td>delete[]</td>
<td>dynamic array deallocation</td>
<td>left to right</td>
</tr>
<tr>
<td>.*</td>
<td>pointer to member via object</td>
<td>right to left</td>
</tr>
<tr>
<td>-&gt;*</td>
<td>pointer to member via pointer</td>
<td>right to left</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>left to right</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>left to right</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>left to right</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>left to right</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>bitwise left shift</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>bitwise right shift</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;</td>
<td>relational less than</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;=</td>
<td>relational less than or equal to</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;</td>
<td>relational greater than</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;=</td>
<td>relational greater than or equal to</td>
<td>left to right</td>
</tr>
<tr>
<td>==</td>
<td>relational is equal to</td>
<td>left to right</td>
</tr>
<tr>
<td>!=</td>
<td>relational is not equal to</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>left to right</td>
</tr>
<tr>
<td>^</td>
<td>bitwise exclusive OR</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise inclusive OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:?</td>
<td>ternary conditional</td>
<td>right to left</td>
</tr>
</tbody>
</table>

Fig. A.2  | C++ operator precedence chart. (Part 2 of 3.)
## Appendix A Operator Precedence Charts

<table>
<thead>
<tr>
<th>C++ Operator</th>
<th>Type</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>assignment</td>
<td>right to left</td>
</tr>
<tr>
<td>+=</td>
<td>addition assignment</td>
<td></td>
</tr>
<tr>
<td>-=</td>
<td>subtraction assignment</td>
<td></td>
</tr>
<tr>
<td>*=</td>
<td>multiplication assignment</td>
<td></td>
</tr>
<tr>
<td>/=</td>
<td>division assignment</td>
<td></td>
</tr>
<tr>
<td>%=</td>
<td>modulus assignment</td>
<td></td>
</tr>
<tr>
<td>&amp;=</td>
<td>bitwise AND assignment</td>
<td></td>
</tr>
<tr>
<td>^=</td>
<td>bitwise exclusive OR assignment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>bitwise inclusive OR assignment</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>bitwise left shift assignment</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>bitwise right shift with sign</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
<td>left to right</td>
</tr>
</tbody>
</table>

**Fig. A.2** | C++ operator precedence chart. (Part 3 of 3.)
The digits at the left of the table are the left digits of the decimal equivalent (0–127) of the character code, and the digits at the top of the table are the right digits of the character code. For example, the character code for “F” is 70, and the character code for “&” is 38.
Here are only numbers ratified.
—William Shakespeare

**Objectives**

In this appendix, you’ll learn:

■ To understand basic number systems concepts such as base, positional value and symbol value.

■ To understand how to work with numbers represented in the binary, octal and hexadecimal number systems.

■ To be able to abbreviate binary numbers as octal numbers or hexadecimal numbers.

■ To be able to convert octal numbers and hexadecimal numbers to binary numbers.

■ To be able to convert back and forth between decimal numbers and their binary, octal and hexadecimal equivalents.

■ To understand binary arithmetic and how negative binary numbers are represented using two’s complement notation.
In this appendix, we introduce the key number systems that programmers use, especially when they are working on software projects that require close interaction with machine-level hardware. Projects like this include operating systems, computer networking software, compilers, database systems and applications requiring high performance.

When we write an integer such as 227 or –63 in a program, the number is assumed to be in the decimal (base 10) number system. The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The lowest digit is 0 and the highest digit is 9—one less than the base of 10. Internally, computers use the binary (base 2) number system. The binary number system has only two digits, namely 0 and 1. Its lowest digit is 0 and its highest digit is 1—one less than the base of 2.

As we’ll see, binary numbers tend to be much longer than their decimal equivalents. Programmers who work in assembly languages and in high-level languages like C that enable programmers to reach down to the machine level, find it cumbersome to work with binary numbers. So two other number systems—the octal number system (base 8) and the hexadecimal number system (base 16)—are popular primarily because they make it convenient to abbreviate binary numbers.

In the octal number system, the digits range from 0 to 7. Because both the binary number system and the octal number system have fewer digits than the decimal number system, their digits are the same as the corresponding digits in decimal.

The hexadecimal number system poses a problem because it requires 16 digits—a lowest digit of 0 and a highest digit with a value equivalent to decimal 15 (one less than the base of 16). By convention, we use the letters A through F to represent the hexadecimal digits corresponding to decimal values 10 through 15. Thus in hexadecimal we can have numbers like 876 consisting solely of decimal-like digits, numbers like 8A55F consisting of digits and letters and numbers like FFE consisting solely of letters. Occasionally, a hexadecimal number spells a common word such as FACE or FEED—this can appear strange to programmers accustomed to working with numbers. The digits of the binary, octal, decimal and hexadecimal number systems are summarized in Figs. C.1–C.2.

Each of these number systems uses positional notation—each position in which a digit is written has a different positional value. For example, in the decimal number 937 (the 9, the 3 and the 7 are referred to as symbol values), we say that the 7 is written in the ones position, the 3 is written in the tens position and the 9 is written in the hundreds position. Each of these positions is a power of the base (base 10) and these powers begin at 0 and increase by 1 as we move left in the number (Fig. C.3).
Appendix C  Number Systems

For longer decimal numbers, the next positions to the left would be the thousands position \(10^3\), the ten-thousands position \(10^4\), the hundred-thousands position \(10^5\), the millions position \(10^6\), the ten-millions position \(10^7\), and so on.

Fig. C.1 | Digits of the binary, octal, decimal and hexadecimal number systems.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Binary</th>
<th>Octal</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>Lowest digit</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Highest digit</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>F</td>
</tr>
</tbody>
</table>

Fig. C.2 | Comparing the binary, octal, decimal and hexadecimal number systems.

<table>
<thead>
<tr>
<th>Positional values in the decimal number system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal digit</td>
</tr>
<tr>
<td>Position name</td>
</tr>
<tr>
<td>Positional value</td>
</tr>
<tr>
<td>Positional value as a power of the base (10)</td>
</tr>
</tbody>
</table>

Fig. C.3 | Positional values in the decimal number system.

For longer decimal numbers, the next positions to the left would be the thousands position \(10^3\), the ten-thousands position \(10^4\), the hundred-thousands position \(10^5\), the millions position \(10^6\), the ten-millions position \(10^7\), and so on.
In the binary number 101, the rightmost 1 is written in the ones position, the 0 is written in the twos position and the leftmost 1 is written in the fours position. Each position is a power of the base (base 2) and these powers begin at 0 and increase by 1 as we move left in the number (Fig. C.4). So, $101 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1 = 5$.

For longer binary numbers, the next positions to the left would be the eights position (2 to the 3rd power), the sixteens position (2 to the 4th power), the thirty-twos position (2 to the 5th power), the sixty-fours position (2 to the 6th power) and so on.

In the octal number 425, we say that the 5 is written in the ones position, the 2 is written in the eights position and the 4 is written in the sixty-fours position. Each of these positions is a power of the base (base 8) and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. C.5).

For longer octal numbers, the next positions to the left would be the five-hundred-and-twelves position (8 to the 3rd power), the four-thousand-and-ninety-sixes position (8 to the 4th power), the thirty-two-thousand-seven-hundred-and-sixty-eights position (8 to the 5th power) and so on.

In the hexadecimal number 3DA, we say that the A is written in the ones position, the D is written in the sixteens position and the 3 is written in the two-hundred-and-fifty-sixes position. Each of these positions is a power of the base (base 16) and these powers begin at 0 and increase by 1 as we move left in the number (Fig. C.6).

For longer hexadecimal numbers, the next positions to the left would be the four-thousand-and-ninety-sixes position (16 to the 3rd power), the sixty-five-thousand-five-hundred-and-thirty-sixes position (16 to the 4th power) and so on.
Appendix C  Number Systems

C.2 Abbreviating Binary Numbers as Octal and Hexadecimal Numbers

The main use for octal and hexadecimal numbers in computing is for abbreviating lengthy binary representations. Figure C.7 highlights the fact that lengthy binary numbers can be expressed concisely in number systems with higher bases than the binary number system.

<table>
<thead>
<tr>
<th>Decimal number</th>
<th>Binary representation</th>
<th>Octal representation</th>
<th>Hexadecimal representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>13</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>14</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>15</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>16</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>17</td>
<td>F</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

Fig. C.7  Decimal, binary, octal and hexadecimal equivalents.

A particularly important relationship that both the octal number system and the hexadecimal number system have to the binary system is that the bases of octal and hexadec-
C.3 Converting Octal and Hexadecimal Numbers to Binary Numbers

In the previous section, we saw how to convert binary numbers to their octal and hexadecimal equivalents by forming groups of binary digits and simply rewriting them as their equivalent octal digit values or hexadecimal digit values. This process may be used in reverse to produce the binary equivalent of a given octal or hexadecimal number.

For example, the octal number 653 is converted to binary simply by writing the 6 as its 3-digit binary equivalent 110, the 5 as its 3-digit binary equivalent 101 and the 3 as its 3-digit binary equivalent 011 to form the 9-digit binary number 110101011.

The hexadecimal number FAD5 is converted to binary simply by writing the F as its 4-digit binary equivalent 1111, the A as its 4-digit binary equivalent 1010, the D as its 4-digit binary equivalent 1101 and the 5 as its 4-digit binary equivalent 0101 to form the 16-digit 1111101011010101.

C.4 Converting from Binary, Octal or Hexadecimal to Decimal

We’re accustomed to working in decimal, and therefore it is often convenient to convert a binary, octal, or hexadecimal number to decimal to get a sense of what the number is “really” worth. Our tables in Section C.1 express the positional values in decimal. To convert a number to decimal from another base, multiply the decimal equivalent of each digit by its positional value and sum these products. For example, the binary number 110101 is converted to decimal 53, as shown in Fig. C.8.
Appendix C  Number Systems

To convert octal 7614 to decimal 3980, we use the same technique, this time using appropriate octal positional values, as shown in Fig. C.9.

To convert hexadecimal AD3B to decimal 44347, we use the same technique, this time using appropriate hexadecimal positional values, as shown in Fig. C.10.

### C.5 Converting from Decimal to Binary, Octal or Hexadecimal

The conversions in Section C.4 follow naturally from the positional notation conventions. Converting from decimal to binary, octal, or hexadecimal also follows these conventions.

Suppose we wish to convert decimal 57 to binary. We begin by writing the positional values of the columns right to left until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

<table>
<thead>
<tr>
<th>Positional values:</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol values:</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Products</td>
<td>1 * 64 = 64</td>
<td>1 * 32 = 32</td>
<td>0 * 16 = 0</td>
<td>1 * 8 = 8</td>
<td>0 * 2 = 0</td>
<td>1 * 1 = 1</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>= 64 + 32 + 0 + 8 + 0 + 1 = 105</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. C.8**  Converting a binary number to decimal.

To convert octal 7614 to decimal 3980, we use the same technique, this time using appropriate octal positional values, as shown in Fig. C.9.

<table>
<thead>
<tr>
<th>Positional values:</th>
<th>512</th>
<th>64</th>
<th>8</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol values:</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Products</td>
<td>7 * 512 = 3584</td>
<td>6 * 64 = 384</td>
<td>1 * 8 = 8</td>
<td>4 * 1 = 4</td>
</tr>
<tr>
<td>Sum</td>
<td>= 3584 + 384 + 8 + 4 = 3980</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. C.9**  Converting an octal number to decimal.

To convert hexadecimal AD3B to decimal 44347, we use the same technique, this time using appropriate hexadecimal positional values, as shown in Fig. C.10.

<table>
<thead>
<tr>
<th>Positional values:</th>
<th>4096</th>
<th>256</th>
<th>16</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol values:</td>
<td>A</td>
<td>D</td>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>Products</td>
<td>A * 4096 = 40960</td>
<td>D * 256 = 3328</td>
<td>3 * 16 = 48</td>
<td>B * 1 = 11</td>
</tr>
<tr>
<td>Sum</td>
<td>= 40960 + 3328 + 48 + 11 = 44347</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. C.10**  Converting a hexadecimal number to decimal.
Then we discard the column with positional value 64, leaving:

<table>
<thead>
<tr>
<th>Positional values:</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol values:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

and thus decimal 57 is equivalent to binary 111001.

To convert decimal 103 to octal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

| Positional values: | 512 | 64 | 8 | 1 |

Then we discard the column with positional value 512, yielding:

| Positional values: | 64 | 8 | 1 |

and thus decimal 103 is equivalent to octal 147.

To convert decimal 375 to hexadecimal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

| Positional values: | 4096 | 256 | 16 | 1 |

Then we discard the column with positional value 4096, yielding:

| Positional values: | 256 | 16 | 1 |

and thus decimal 375 is equivalent to hexadecimal 177.
C.6 Negative Binary Numbers: Two’s Complement Notation

The discussion so far in this appendix has focused on positive numbers. In this section, we explain how computers represent negative numbers using two’s complement notation. First we explain how the two’s complement of a binary number is formed, then we show why it represents the negative value of the given binary number.

Consider a machine with 32-bit integers. Suppose

```
int value = 13;
```

The 32-bit representation of value is

```
00000000 00000000 00000000 00001101
```

To form the negative of value we first form its one’s complement by applying C’s bitwise complement operator (~):

```
onesComplementOfValue = ~value;
```

Internally, ~value is now value with each of its bits reversed—ones become zeros and zeros become ones, as follows:

```
value:
00000000 00000000 00000000 00001101
~value (i.e., value’s ones complement):
11111111 11111111 11111111 11110010
```

To form the two’s complement of value, we simply add 1 to value’s one’s complement. Thus

```
Two’s complement of value:
11111111 11111111 11111111 11110011
```

Now if this is in fact equal to –13, we should be able to add it to binary 13 and obtain a result of 0. Let’s try this:

```
00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
------------------------------------
00000000 00000000 00000000 00000000
```

The carry bit coming out of the leftmost column is discarded and we indeed get 0 as a result. If we add the one’s complement of a number to the number, the result would be all 1s. The key to getting a result of all zeros is that the twos complement is one more than the one’s complement. The addition of 1 causes each column to add to 0 with a carry of 1. The carry keeps moving leftward until it is discarded from the leftmost bit, and thus the resulting number is all zeros.

Computers actually perform a subtraction, such as

```
x = a - value;
```

by adding the two’s complement of value to a, as follows:

```
x = a + (~value + 1);
```
Suppose \( a \) is 27 and \( \text{value} \) is 13 as before. If the two’s complement of \( \text{value} \) is actually the negative of \( \text{value} \), then adding the two’s complement of \( \text{value} \) to \( a \) should produce the result 14. Let’s try this:

\[
\begin{array}{c|c}
 a \text{ (i.e., 27)} & 00000000 00000000 00000000 00011011 \\
 +(-\text{value} + 1) & +11111111 11111111 11111111 11110011 \\
\hline
 & 00000000 00000000 00000000 00001110
\end{array}
\]

which is indeed equal to 14.

**Summary**

- An integer such as 19 or 227 or –63 in a program is assumed to be in the decimal (base 10) number system. The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The lowest digit is 0 and the highest digit is 9—one less than the base of 10.

- Internally, computers use the binary (base 2) number system. The binary number system has only two digits, namely 0 and 1. Its lowest digit is 0 and its highest digit is 1—one less than the base of 2.

- The octal number system (base 8) and the hexadecimal number system (base 16) are popular primarily because they make it convenient to abbreviate binary numbers.

- The digits of the octal number system range from 0 to 7.

- The hexadecimal number system poses a problem because it requires 16 digits—a lowest digit of 0 and a highest digit with a value equivalent to decimal 15 (one less than the base of 16). By convention, we use the letters A through F to represent the hexadecimal digits corresponding to decimal values 10 through 15.

- Each number system uses positional notation—each position in which a digit is written has a different positional value.

- A particularly important relationship of both the octal number system and the hexadecimal number system to the binary system is that the bases of octal and hexadecimal (8 and 16 respectively) are powers of the base of the binary number system (base 2).

- To convert an octal to a binary number, replace each octal digit with its three-digit binary equivalent.

- To convert a hexadecimal number to a binary number, simply replace each hexadecimal digit with its four-digit binary equivalent.

- Because we’re accustomed to working in decimal, it is convenient to convert a binary, octal or hexadecimal number to decimal to get a sense of the number’s “real” worth.

- To convert a number to decimal from another base, multiply the decimal equivalent of each digit by its positional value and sum the products.

- Computers represent negative numbers using two’s complement notation.

- To form the negative of a value in binary, first form its one’s complement by applying C’s bitwise complement operator (~). This reverses the bits of the value. To form the two’s complement of a value, simply add one to the value’s one’s complement.
Appendix C  Number Systems

base 16 number system 925
binary number system 925
decimal number system 925
digit 925
hexadecimal number system 925

negative value 932
octal number system 925
one’s complement notation 932
position notation 925
position value 925
symbol value 925
two’s complement notation 932

Self-Review Exercises

C.1  Fill in the blanks in each of the following statements:
   a) The bases of the decimal, binary, octal and hexadecimal number systems are ________, ________, ________ and ________ respectively.
   b) The positional value of the rightmost digit of any number in either binary, octal, decimal or hexadecimal is always ________.
   c) The positional value of the digit to the left of the rightmost digit of any number in binary, octal, decimal or hexadecimal is always equal to ________.

C.2  State whether each of the following is true or false. If false, explain why.
   a) A popular reason for using the decimal number system is that it forms a convenient notation for abbreviating binary numbers simply by substituting one decimal digit per group of four binary bits.
   b) The highest digit in any base is one more than the base.
   c) The lowest digit in any base is one less than the base.

C.3  In general, the decimal, octal and hexadecimal representations of a given binary number contain (more/fewer) digits than the binary number contains.

C.4  The (octal / hexadecimal / decimal) representation of a large binary value is the most concise (of the given alternatives).

C.5  Fill in the missing values in this chart of positional values for the rightmost four positions in each of the indicated number systems:

<table>
<thead>
<tr>
<th></th>
<th>decimal</th>
<th>1000</th>
<th>100</th>
<th>10</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>hexadecimal</td>
<td>...</td>
<td>256</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>binary</td>
<td>...</td>
<td>512</td>
<td>...</td>
<td>8</td>
<td>...</td>
</tr>
</tbody>
</table>

C.6  Convert binary 110101011000 to octal and to hexadecimal.

C.7  Convert hexadecimal FACE to binary.

C.8  Convert octal 7316 to binary.

C.9  Convert hexadecimal 4FEC to octal. [Hint: First convert 4FEC to binary, then convert that binary number to octal.]

C.10 Convert binary 1101110 to decimal.

C.11 Convert octal 317 to decimal.

C.12 Convert hexadecimal EFD4 to decimal.

C.13 Convert decimal 177 to octal, to octal and to hexadecimal.

C.14 Show the binary representation of decimal 417. Then show the one’s complement of 417 and the two’s complement of 417.

C.15 What is the result when a number and its two’s complement are added to each other?
Answers to Self-Review Exercises

C.1  a) 10, 2, 8, 16.  b) 1 (the base raised to the zero power).  c) The base of the number system.

C.2  a) False. Hexadecimal does this.  b) False. The highest digit in any base is one less than the base.  c) False. The lowest digit in any base is zero.

C.3  Fewer.

C.4  Hexadecimal.

C.5  

<table>
<thead>
<tr>
<th>Base</th>
<th>1000</th>
<th>100</th>
<th>10</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>4096</td>
<td>256</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Binary</td>
<td>512</td>
<td>64</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

C.6  Octal 6530; Hexadecimal D58.

C.7  Binary 1111 1010 1100 1110.

C.8  Binary 111 011 001 110.

C.9  Binary 0100 1111 1101 100; Octal 47754.

C.10  Decimal 2+4+8+32+64=110.

C.11  Decimal 7+1*8+3*64=7+8+192=207.

C.12  Decimal 4+13*16+15*256+14*4096=61396.

C.13  Decimal 177

to binary:

\[
\begin{array}{cccccccc}
256 & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\end{array}
\]

\[(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)\]

10110001


to octal:

\[
\begin{array}{cccc}
512 & 64 & 8 & 1 \\
64 & 8 & 1 \\
(2*64)+(6*8)+(1*1) \\
261 \\
\end{array}
\]


to hexadecimal:

\[
\begin{array}{cccc}
256 & 16 & 1 \\
16 & 1 \\
(11*16)+(1*1) \\
(B*16)+(1*1) \\
B1 \\
\end{array}
\]

C.14  Binary:

\[
\begin{array}{cccccccccccc}
512 & 256 & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
256 & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\end{array}
\]

\[(1*256)+(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)\]

110100001

One’s complement: 001011110
Two’s complement: 001011111
Check: Original binary number + its two’s complement
Appendix C  Number Systems

110100001
001011111
---------
000000000
C.15 Zero.

Exercises

C.16 Some people argue that many of our calculations would be easier in the base 12 number system because 12 is divisible by so many more numbers than 10 (for base 10). What is the lowest digit in base 12? What would be the highest symbol for the digit in base 12? What are the positional values of the rightmost four positions of any number in the base 12 number system?

C.17 Complete the following chart of positional values for the rightmost four positions in each of the indicated number systems:

<table>
<thead>
<tr>
<th></th>
<th>decimal</th>
<th>base 6</th>
<th>base 13</th>
<th>base 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1000</td>
<td>...</td>
<td>169</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>6</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

C.18 Convert binary 100101111010 to octal and to hexadecimal.

C.19 Convert hexadecimal 3A7D to binary.

C.20 Convert hexadecimal 765F to octal. (Hint: First convert 765F to binary, then convert that binary number to octal.)

C.21 Convert binary 1011110 to decimal.

C.22 Convert octal 426 to decimal.

C.23 Convert hexadecimal FFFF to decimal.

C.24 Convert decimal 299 to binary, to octal and to hexadecimal.

C.25 Show the binary representation of decimal 779. Then show the one’s complement of 779 and the two’s complement of 779.

C.26 Show the two’s complement of integer value –1 on a machine with 32-bit integers.
D.1 Introduction

The game of Sudoku exploded in popularity worldwide in 2005. Almost every major newspaper now publishes a Sudoku puzzle daily. Handheld game players let you play anytime, anywhere and create puzzles on demand at various levels of difficulty.

A completed Sudoku puzzle is a 9×9 grid (i.e., a two-dimensional array) in which the digits 1 through 9 appear once and only once in each row, each column and each of nine 3×3 grids. In the partially completed 9×9 grid of Fig. D.1, row 1, column 1, and the 3×3 grid in the upper-left corner of the board each contain the digits 1 through 9 once and only once. We use C’s two-dimensional array row and column-numbering conventions, but we’re ignoring row 0 and column 0 in conformance with Sudoku community conventions.

![Fig. D.1](image)

Partially completed 9×9 Sudoku grid. Note the nine 3×3 grids.

The typical Sudoku puzzle provides many filled-in cells and many blanks, often arranged in a symmetrical pattern as is typical with crossword puzzles. The player’s task is to fill in the blanks to complete the puzzle. Some puzzles are easy to solve; some are quite difficult, requiring sophisticated solution strategies.

We’ll discuss various simple solution strategies, and suggest what to do when these fail. We’ll also present approaches for programming Sudoku puzzle creators and solvers in C. Unfortunately, Standard C does not include graphics and GUI (graphical user interface) capabilities, so our repre-
sentation of the board won’t be as elegant as we could make it in Java and other programming languages that support these capabilities. You may want to revisit your Sudoku programs after you read Appendix E, Game Programming with the Allegro C Library. Allegro, which is not part of Standard C, offers capabilities that will help you add graphics and even sounds to your Sudoku programs.

D.2 Deitel Sudoku Resource Center

Check out our Sudoku Resource Center at www.deitel.com/sudoku. It contains downloads, tutorials, books, e-books and more that will help you master the game. Trace the history of Sudoku from its origin in the eighth century through modern times. Download free Sudoku puzzles at various levels of difficulty, enter daily game contests to win Sudoku books, and get a daily Sudoku puzzle to post on your web site. Get great beginner’s resources—learn the rules of Sudoku, receive hints on solving sample puzzles, learn the best solution strategies and get free Sudoku solvers—just type in the puzzle from your newspaper or favorite Sudoku site and get an immediate solution; some Sudoku solvers even provide detailed step-by-step explanations. Get mobile device Sudoku games that can be installed on cell phones, Palm® devices, Game Boy® players and Java-enabled devices. Some Sudoku sites have timers, signal when an incorrect number is placed and provide hints. Purchase T-shirts and coffee mugs with Sudoku puzzles on them, participate in Sudoku player forums, get blank Sudoku worksheets that can be printed and check out hand-held Sudoku game players—one offers a million puzzles at five levels of difficulty. Download free Sudoku puzzle maker software. And not for the faint of heart—try fiendishly difficult Sudokus with tricky twists, a circular Sudoku and a variant of the puzzle with five interlocking grids. Subscribe to our free newsletter, the Deitel® Buzz Online, for notifications of updates to our Sudoku Resource Center and to other Deitel Resource Centers at www.deitel.com that provide games, puzzles and other interesting programming projects.

D.3 Solution Strategies

When we refer to a Sudoku 9×9 grid, we’ll call it array s. By looking at all the filled-in cells in the row, column and 3×3 grid that includes a particular empty cell, the value for that cell might become obvious. Trivially, cell s[1][7] in Fig. D.2 must be 6.

![Fig. D.2](image_url) | Determining the value of a cell by checking all filled-in cells in the same row.
Less trivially, to determine the value of \( s[1][7] \) in Fig. D.3, you have to pick up hints from row 1 (i.e., the digits 3, 6 and 9 are taken), column 7 (i.e., the digits 4, 7 and 1 are taken) and the upper-right \( 3 \times 3 \) grid (i.e., the digits 9, 8, 4 and 2 are taken). Here the empty cell \( s[1][7] \) must be 5—the only number not already mentioned in row 1, column 7 or the upper-right \( 3 \times 3 \) grid.

**Fig. D.3**  |  Determining the value of a cell by checking all filled-in cells in the same row, column and \( 3 \times 3 \) grid.

### Singletons

The strategies we’ve discussed so far can easily determine the final digits for some open cells, but you’ll often have to dig deeper. Column 6 of Fig. D.4 shows cells with already determined values (e.g., \( s[1][6] \) is a 9, \( s[3][6] \) is a 6, etc.), and cells indicating the set of values (which we call “possibles”) that at this point are still possible for that cell.

Cell \( s[6][6] \) contains 257, indicating that only the values 2, 5 or 7 can eventually be assigned to this cell. The other two open cells in column 6—\( s[2][6] \) and \( s[5][6] \)—are both 27, indicating that only the values 2 or 7 can eventually be assigned to these cells. Thus \( s[6][6] \), the only cell in column 6 that lists 5 as a remaining possible value, must be 5. We call that value 5 a singleton. So we can commit cell \( s[6][6] \) to a 5 (Fig. D.5), somewhat simplifying the puzzle.

### Doubles

Consider the upper-right \( 3 \times 3 \) grid in Fig. D.6. The dashed cells could already be committed or could have lists of possible values. Notice the doubles—the two cells \( s[1][9] \) and \( s[2][7] \) containing only the two possibilities 15. If \( s[1][9] \) ultimately becomes 1, then \( s[2][7] \) must be 5; if \( s[1][9] \) ultimately becomes 5, then \( s[2][7] \) must be 1. So between them, those two cells will definitely “use up” the 1 and the 5. Thus 1 and 5 can be eliminated from cell \( s[3][9] \) that contains the possible values 1357, so we can rewrite its contents as 37, simplifying the puzzle a bit. If cell \( s[3][9] \) had originally contained only 135, then eliminating the 1 and the 5 would enable us to force the cell to the value 3.

Doubles can be more subtle. For example, suppose two cells of a row, column or \( 3 \times 3 \) grid have possibles lists of 2467 and 257 and that no other cell in that row, column or \( 3 \times 3 \) grid mentions 2 or 7 as a possible value. Then, 27 is a hidden double—one of those two cells must be 2 and the other must be 7, so all digits other than 2 and 7 can be removed from the possibles lists of those
two cells (i.e., 2467 becomes 27 and 257 becomes 27—creating a pair of doubles—thus somewhat simplifying the puzzle).

---

**Fig. D.4** | Notation showing the complete sets of possible values for open cells.

---

**Fig. D.5** | Committing cell s[6][6] to the singleton value 5.
Consider column 5 of Fig. D.7. The dashed cells could already be committed or could have lists of possible values. Notice the triples—the three cells containing the exact same three possibilities 467, namely cells s[1][5], s[6][5] and s[9][5]. If one of those three cells ultimately becomes 4, then
the others reduce to doubles of 67; if one of those three cells ultimately becomes 6, then the others reduce to doubles of 47; and if one of those three cells ultimately becomes 7, then the others reduce to doubles of 46. Among the three cells containing 467, one must ultimately be 4, one must be 6, and one must be 7. Thus the 4, 6 and 7 can be eliminated from cell \( s[4][5] \) that contains the possibilities 14567, so we can rewrite its contents as 15, simplifying the puzzle a bit. If cell \( s[4][5] \) had originally contained 1467, then eliminating the 4, 6 and 7 would enable us to force the value 1 in that cell.

Triples can be more subtle. Suppose a row, column or \( 3 \times 3 \) grid contains cells with possibilities lists of 467, 46, and 67. Clearly one of those cells must be 4, one must be 6 and one must be 7. Thus 4, 6 and 7 can be removed from all the other possibilities lists in that row, column or \( 3 \times 3 \) grid.

Triples can also be hidden. Suppose that a row, column or \( 3 \times 3 \) grid contains the possibilities lists 5789, 259 and 13789, and that no other cell in that row, column or \( 3 \times 3 \) grid mentions 5, 7 or 9. Then one of those cells must be 5, one must be 7 and one must be 9. We call 579 a hidden triple and all possibilities other than 5, 7 and 9 can be deleted from those three cells (i.e., 5789 becomes 579, 259 becomes 59 and 13789 becomes 79), thus somewhat simplifying the puzzle.

Other Sudoku Solution Strategies

There are a number of other Sudoku solution strategies. Here are two of the many sites we recommend in our Sudoku Resource Center (www.deitel.com/sudoku) that will help you dig deeper:

www.sudokuoftheday.com/pages/techniques-overview.php
www.angusj.com/sudoku/hints.php

D.4 Programming Sudoku Puzzle Solvers

In this section we suggest how to program Sudoku solvers. We use a variety of approaches. Some may seem unintelligent, but if they can solve Sudokus faster than any human on the planet, then perhaps they are in some sense intelligent.

If you've done our Knight's Tour exercises (Exercises 6.24, 6.25 and 6.29) and Eight Queens exercises (Exercises 6.26 and 6.27), you've implemented various brute force and heuristic problem-solving approaches. In the next several sections, we suggest brute force and heuristic Sudoku-solving strategies. You should try programming them, as well as creating and programming your own. Our goal is simply to acquaint you with Sudoku, and some of its challenges and problem-solving strategies. Along the way, you'll become more facile with manipulating two-dimensional arrays and with nested iteration structures. We have made no attempt to produce optimal strategies, so once you analyze our strategies, you'll want to consider how you can improve upon them.

Programming a Solution for “Easy” Sudokus

The strategies we’ve shown—eliminating possibilities based on values already committed in a cell’s row, column and \( 3 \times 3 \) grid; and simplifying a puzzle using singletons, doubles (and hidden doubles) and triples (and hidden triples)—are sometimes sufficient to solve a puzzle. You can program the strategies then iterate on them until all 81 squares are filled. To confirm that the filled puzzle is a valid Sudoku, you can write a function to check that each row, column and \( 3 \times 3 \) grid contains the digits 1 through 9 once and only once. Your program should apply the strategies in order. Each of them either forces a digit in a cell or simplifies the puzzle a bit. When any one of the strategies works, return to the beginning of your loop and reapply the strategies in order. When a strategy doesn't work, try the next. For “easy” Sudokus, these techniques should generate a solution.

Programming a Solution for Harder Sudokus

For harder Sudokus, your program will eventually reach a point where it still has uncommitted cells with possibilities lists, and none of the simple strategies we’ve discussed will work. If this hap-
pens, first save the state of the board, then generate the next move by randomly choosing one of the possible values in any of the remaining cells. Then reevaluate the board, enumerating the remaining possibilities for each cell. Then try the basic strategies again, looping through them repeatedly, until either the Sudoku is solved, or the strategies once again no longer improve the board, at which point you can again try another move at random. If you reach a point where there are still empty cells, but no possible digits for at least one of those cells, the program should abandon that attempt, restore the board state that you saved, and begin the random approach again. Keep looping until a solution is found.

D.5 Generating New Sudoku Puzzles

First, let’s consider approaches for generating valid finished 9×9 Sudokus with all 81 squares filled in. Then, we’ll suggest how to empty some cells to create puzzles that people can attempt.

Brute Force Approaches

When personal computers appeared in the late 1970s, they processed tens of thousands of instructions per second. Today’s desktop computers commonly process billions of instructions per second and the world’s fastest supercomputers can process trillions of instructions per second! Brute force approaches that might have required months of computing in the 1970s can now produce solutions in seconds! This encourages people who need results quickly to program simple brute force approaches and get solutions sooner than by taking the time to develop more sophisticated “intelligent” problem solving strategies. Although our brute force approaches may seem ponderous, they will mechanically grind out solutions.

For these approaches you’ll need some utility functions. Create the function

```c
int validSudoku( int sudokuBoard[10][10] );
```

which receives a Sudoku board as a two-dimensional array of integers (recall that we’re ignoring row 0 and column 0). This function should return 1 if a completed board is valid, 2 if a partially completed board is valid and 0 otherwise.

An Exhaustive Brute Force Approach

One brute force approach is simply to select all possible placements of the digits 1 through 9 in every cell. This could be done with 81 nested for statements that each loop from 1 through 9. The number of possibilities (981) is so vast that you might say it’s not worth trying. But this approach does have the advantage that it will eventually stumble onto every possible solution, some of which could show up fortuitously early on.

A slightly more intelligent version of this exhaustive brute-force approach would be to check each digit you’re about to place to see if it leaves the board in a valid state. If it does, then move on to placing a digit in the next cell. If the digit you’re attempting to place leaves the board in an invalid state, then try all other eight digits on that cell in order. If one of them works, then move on to the next cell. If none of them works, then move back up to the previous cell and try its next value. Nested for statements can handle this automatically.

Brute Force Approach with Randomly Selected Row Permutations

Every row, column, and 3×3 grid on a valid Sudoku contains a permutation of the digits 1 through 9. There are 9! (i.e., 9·8·7·6·5·4·3·2·1 = 362,880) such permutations. Write a function

```c
void permutations( int sudokuBoard[10][10] );
```

that receives a 10×10 two-dimensional array and in the 9×9 portion of it that corresponds to a Sudoku grid fills each of the nine rows with a randomly selected permutation of the digits 1 through 9.
Here's one way to generate a random permutation of the digits 1 through 9—for the first digit, simply choose a random digit from 1 through 9; for the second digit, use a loop to repeatedly generate a random digit from 1 through 9 until a digit different from the first digit is selected; for the third digit, use a loop to repeatedly generate a random digit from 1 through 9 until a digit different from the first two digits is selected; and so on.

After placing nine randomly selected permutations into the nine rows of your Sudoku array, run function validSudoku on the array. If it returns 1, you're done. If it returns 0, simply loop again, generating another nine randomly selected permutations of the digits 1 through 9 into the nine successive rows of the array Sudoku. The simple process will generate valid Sudokus. By the way, this approach guarantees that all the rows are valid permutations of the digits 1 through 9, so you should add an option to your function validSudoku that will have it check only columns and 3x3 grids.

**Heuristic Solution Strategies**

When we studied the Knight’s Tour in Exercises 6.24, 6.25 and 6.29, we developed a “keep your options open” heuristic. To review, a heuristic is a guideline. It “sounds good” and seems like a reasonable rule to follow. It’s programmable, so it gives us a way to direct a computer to attempt to solve a problem. But heuristic approaches don’t necessarily guarantee success. For complex problems like solving a Sudoku puzzle, the number of possible placements of the digits 1–9 is enormous, so the hope in using a reasonable heuristic is that it will avoid wasting time on fruitless possibilities and instead focus on solution attempts much more likely to yield success.

**A “Keep Your Options Open” Sudoku-Solving Heuristic**

Let’s develop a “keep your options open” heuristic for solving Sudokus. At any point in solving a Sudoku, we can categorize the board by listing in each empty cell the digits from 1 to 9 which are still open possibilities for that cell. For example, if a cell contains 3578, then the cell must eventually become 3, 5, 7 or 8. When attempting to solve a Sudoku, we reach a dead end when the number of possible digits that can be placed in an empty cell becomes zero. So, consider the following strategy:

1. Associate with every empty square a possibles list of the digits that can still be placed in that square.
2. Characterize the state of the board by simply counting the number of possible placements for the entire board.
3. For each possible placement for each empty cell, associate with that placement the count that would characterize the state of the board after that placement.
4. Then, place the particular digit in the particular empty square (of all those that remain) that leaves the board count the highest (in case of a tie, pick one at random). This is a key to “keeping your options open.”

**Lookahead Heuristic**

This is simply an embellishment of our “keep your options open” heuristic. In case of a tie, look ahead one more placement. Place the particular digit in the particular square whose subsequent placement leaves the board count the highest after two moves out.

**Forming Sudoku Puzzles with Empty Cells**

Once you get your Sudoku generator program running, you should be able to generate lots of valid Sudokus quickly. To form a puzzle, save the solved grid, then empty some cells. One way to do this is to empty randomly chosen cells. A general observation is that Sudokus tend to become more difficult as the empty cells increase (there are exceptions to this).
Another approach is to empty the cells in a manner that leaves the resulting board symmetric. This can be done programmatically by randomly picking a cell to empty, then emptying its “reflecting cell.” For example, if you empty the top-left cell $s[1][1]$, you might empty the bottom-left cell $s[9][1]$ as well. Such reflections are calculated by presenting the column, but determining the row by subtracting the initial row from 10. You could also do the reflections by subtracting both the row and column of the cell you’re emptying from 10. Hence, the reflecting cell to $s[1][1]$ would be $s[10-1][10-1]$ or $s[9][9]$.

A Programming Challenge

Published Sudoku puzzles typically have exactly one solution, but it’s still satisfying to solve any Sudoku, even ones that have multiple solutions. Develop a means of demonstrating that a particular Sudoku puzzle has exactly one solution.

D.6 Conclusion

This appendix on solving and programming Sudoku puzzles has presented you with many challenges. Be sure to check out our Sudoku Resource Center (www.deitel.com/sudoku/) for numerous web resources that will help you master Sudoku and develop various approaches for writing programs to create and solve existing Sudoku puzzles.
Appendices on the Web

The following appendices are available as PDF documents from this book’s Companion Website (www.pearsonhighered.com/deitel/):

• Appendix E, Game Programming with the Allegro C Library
• Appendix F, Sorting: A Deeper Look
• Appendix G, Introduction to C99
• Appendix H, Using the Visual Studio Debugger
• Appendix I, Using the GNU Debugger

These files can be viewed in Adobe® Reader® (get.adobe.com/reader). The index entries for these appendices have uppercase Roman numeral page numbers.

New copies of this book come with a Companion Website access code that is located on the card inside the book’s front cover. If the access code is already visible or there is no card, you purchased a used book or an edition that does not come with an access code. In this case, you can purchase access directly from the Companion Website.
Symbols
\t horizontal-tab escape sequence 26
\b bitwise exclusive OR operator 373, 395
\a bitwise exclusive OR assignment operator 402
__func__ predefined identifier XCVIII
__VA_ARGS__ XCV
Pragma operator XCV
:: (binary scope resolution operator) 586, 660
::: unary scope resolution operator 544
!= inequality operator 38, 39, 676
& bitwise AND operator 386
. dot operator 386
. structure member operator 386
.h filename extension 580
" 368
"w" file open mode 422
* assignment suppression character 374
* multiplication operator 34, 72
*= multiplication assignment operator 80
/ division operator 72
// single-line comment 25, 530, LXX
/= division assignment operator 80
\ backlash-character escape sequence 26
\? escape sequence 369
\' single-quote-character escape sequence 369
" double-quote-character escape sequence 369
\" backlash-character escape sequence 369
\0 null character escape sequence 207
\a alert escape sequence 26, 369
\b escape sequence 369
\f form-feed escape sequence 369
\n escape sequence 312
\n newline escape sequence 26, 369
\r carriage-return escape sequence 369
\r escape sequence 312
\t escape sequence 312
\t horizontal-tab escape sequence 369
\v escape sequence 312, 369
\& address operator 31
\& and * pointer operators 256
\& bitwise AND operator 395
\& to declare reference 538
\& operator 116, 119, 173
\& bitwise AND assignment operator 402
#define 496
#endif 499
#error 500
#error 500
if 499
pragma 500
#undef 501
% character in a conversion specifier 72, 357, 362
% remainder operator 34, 34, 153
% conversion specifier 363
%e conversion specifier 150, 361, 372
%d conversion specifier 150
%e conversion specifier 360, 371
%f conversion specifier 72, 150
%g conversion specifier 371
%h conversion specifier 150
%hu conversion specifier 150
%i conversion specifier 371
%id conversion specifier 150
%i conversion specifier 150
%lu conversion specifier 150
%n conversion specifier 362
%p conversion specifier 256, 362
%p conversion specifier 283, 361, 372
%u conversion specifier 150, 359
%x conversion specifier 370
%X conversion specifier 370
+ flag 367
+ flag 366
- minus operator 80
+ unary plus operator 80
- operator 78, 80, 274
+ operator 78, 79, 80, 274
+= addition assignment operator 77, 80, 676
< less than operator 38
< redirection input symbol 508
<< left-shift operator 395
<< stream insertion operator 532
<< left-shift assignment operator 402
<< less-than-or-equal-to operator 38
<< assignment operator 80
<< subtraction assignment operator 80
<< equality operator 38, 120, 676
>> greater than operator 38
>> redirection output symbol 509
>> structure pointer operator 386
>> greater-than-or-equal-to operator 38
>> append output symbol 509
>> right-shift operator 395
>>> right-shift assignment operator 402
|= bitwise inclusive OR operator 402
| | 174
~ bitwise one’s complement 395
~, bitwise complement operator 400

Numerics
0 Conversion specifier 30, 31, 370, 371
0X 868
0x 367, 868

A
a file open mode 425
a.out 14
ab file open mode 425
ab binary file open mode 516
ab file open mode 425
ab+ binary file open mode 516
ab+ file open mode 425
abnormal program termination 518
abort a program 911
abort function 502, 502, 621, 901, 906
absolute-value 143
abstract base class 780, 799, 800
abstract class 799, 800, 801, 819
abstract data type (ADT) 664
abstraction 144
access function 612
access global variable 544
access non-static class data members
and member functions 663
access private member of a class 572
access privileges 33, 262
access specifier 564, 571, 652
private 571
protected 603
public 564, 571
access-specifier label
private: 571
public: 564
access the caller’s data 537
access violation 33, 312, 361
accessibility heuristic 249
accessor 574
accounts receivable 135
accumulated outputs 331
accumulator 302, 304, 306
action 26, 27, 38, 56, 63, 664
action oriented 552
action statement 56
action symbol 57
action/decision model 27, 59
actions 38, 55
actions (computers perform) 2

Index
global 586
    global function 845
    global function to overload an operator 677
    global namespace scope 660, 846
    global object constructors 621
    global scope 622
    global variable 162, 163, 164, 270, 392, 512, 544
    Global Warming Facts Quiz 193
    global, friend function 681
    global, non-friend function 675
    GNU GCC 4.3 xxvii
    hardware platform
    hardware independent 7
    head of a queue
    has-a
    hardware registers 162
    hardware
    hardcopy printer 14
    hard disk 4, 12
    halt 306
    halt instruction 304
    handle on an object 609
    hard disk 4, 12
    hardcopy printer 14
    hardware 2, 4, 6
    hardware independent 7
    hardware platform 7
    hardware registers 162
    has-a relationship 729, 645
    head of a queue 455, 472
    header 25, 151, 152, 496, 535
    header file 152, 533, 580, 588, 604, 772, 822
    <ctype.h> 312
    <exception> 534, 892
    <fstream> 534
    <functional> 534
    <iostream> 533
    <iterator> 534
    <limits> 534
    <list> 533
    <locale> 534
    <memory> 534, 907
    <queue> 533
    <set> 533
    <sstream> 534
    <stack> 533
    <stde decltype> 534, 892, 910
    <string> 534, 567
    <typeinfo> 534, 825
    Allegro datatime XLVI, XLVII
    complex.h XC
    fenv.h LXXX
    header files (cont.)
    inttypes.h LXXX
    iso646, h LXXX
    location 583
    name enclosed in angle brackets (< >) 583
    name enclosed in quotes (" ") 583
    stdbool.h LXXXX
    stdint.h LXXX
    tcmath.h LXXX
    wchar.h LXXX
    wctype.h LXXX
    heap 682
    help debugger command CXXI
    helper function 612
    heuristic 248
    heuristic problem solving approach 942, 944
    hex stream manipulator 863, 868, 872
    hexadecimal 137, 313, 320, 358, 363
    hexadecimal (base 16) number system 925
    hexadecimal (base-16) number 857, 863, 868, 872
    hexadecimal integer 256
    hexadecimal notation 857
    hide an internal data representation 665
    hide implementation details 611
    hide private data from clients 611
    hierarchical boss function/worker
    function relationship 142
    hierarchy of exception classes 909
    hierarchy of shapes 799
    high-level language 7
    highest level of precedence 35
    high-level I/O 853
    High-performance card shuffling and
dealing simulation 389
    histogram 135, 205
    Histogram printing 205
    history of Sudoku 938
    horizontal tab (\t) 26, 312
    host object 646
    HourlyEmployee class header file 809
    HourlyEmployee class implementation
    file 810
    Huge integers 725
    HugeInt class 722
    HugeInteger Class exercise 634
    hypotenuse of a right triangle 187
    IBM Corporation 5, 10
    IBM Personal Computer 5
    identifier(s) 29, 497
    if selection statement 38, 59, 62
    if statement 38
    if...else selection statement 57, 59, 74
    ifndef preprocessor directive 499
    ifndef preprocessor directive 604
    ifelse selection statement 60
    ifndef preprocessor directive 499
    #ifndef 680
    ignore function of istream 861
    illegal instruction 518
    image 14
    Implementation class definition 712
    implementation file 713
    implementation inheritance 802
    implementation of a member function
    changes 618
    Implementing a proxy class 714
    implicit conversion 71, 698, 708, 709, 711
    via conversion constructors 709
    implicit first argument 654
    implicit handle 609
    implicit int LXXXIX
    implicit, user-defined conversions 698
    implicitly virtual 793
    Importing and Playing Sounds in Allegro
    XVI
    Importing Bitmaps in Allegro IV
    importing fonts in Allegro XXVI
    improper implicit conversion 708
    INCITS/ISO/IEC 9899-1999 (C standard document) 8
    #include preprocessor directive 202, 496, 535
    including a header file multiple times 604
    including headers 151
    increment 103
    increment a control variable 99
    increment a pointer 273
    increment of a control variable 102, 104
    increment operator (++) 78
    incremented 274
    indefinite postponement 281, 297
    indefinite repetition 67, 98
    indent 27
    indention 58, 60, 62
    independent software vendor (ISV) 532,
    611, 771, 822
    index (or subscript) 196
    indirect base class 728, 731
    indirect derived class 813
    indirect 254, 258
    indirect operator (*) 152, 256, 258
    indirectly reference a value 254
    inequality operator (!=) 685
    infinite loop 63, 71, 102
    infinite recursion 170, 693
    infix notation 489
    infix-to-postfix conversion 489
    infobreak debugger command CXXIII
    information hiding 164, 267, 552, 664
    inherit implementation 830
    inherit interface 799, 830
    inherit members of an existing class 728
    inheritance 551, 603, 609, 728, 731,
    771, 822, 833
    implementation vs. interface
    inheritance 802
    Inheritance Advantage 775
    inheritance examples 730
    inheritance hierarchy 793, 801
    Inheritance hierarchy for university
    CommunityMembers 730
    inheritance relationships of I/O-related
    classes 856
    inheriting interface versus inheriting
    implementation 830
    initial value of a control variable 99, 104
    initialization phase 69
    initialize a constant of a built-in data type
    641
    ...
left side of an assignment 624, 692
left stream manipulator 869
left stream manipulator 868, 869
left subtree 478
left-shift operator (<) 394, 414, 673, 855
legacy code 262
length member function of class string 590
length modifier 359
length of a substring 698
length_error exception 510
letters 418
level order binary tree traversal 483, 493, 495
library function 8
LIFO (last-in, first-out) 151
order 837, 842
LIFO (last-in-first-out) 466
limerick exercise 349
lines
line of text 501
local area network (LAN)
local automatic object 623
local scope 609
local variable 144, 162, 163, 209, 569
Local Variable Destructors 917
locale 152
<stdio.h> header file 534
<stdio.h> header file 152
Locals window CX
Locals window (Visual C++ 2005 debugger) CX
location 33
log function 143
log10 function 143
log2 comparisons 483
logic error 39, 63, 66, 101, 120, 149, 202, 392, CV, CXIX
logic_error exception 910
logical AND operator (&&) 116, 116, 397
logical decision 3
logical negation (NOT) operator (!) 117, 118
logical OR operator (||) 116, 399
logical page 369
logical unit 4
Logo language 246
long 113, 168
long double 150, 516
long int 150, 168, 516
long integer 516
long long int
lookahead heuristic for solving Sudoku 944
loop 67, 101
loop continuation condition 101, 102, 103, 113
loop counter 102
loop-continuation condition 98
looping 101
Lord Byron 10
loss of data 878
Lovelace, Ada 10
lowercase letter 51, 152, 534
low-level I/O capabilities 853
left ("left value")
member function
member 383
member function 552, 562
member selection operator (.) 562, 673
member initializer 641, 643, 694
member initializer for a const data member 643
member-initializer list 642, 646, 648, 649
member-initializer syntax 641
member name (bit field) 403
member object destructors 917
member-object initializer 650
member object's default constructor 651
member-initializer list 642, 646, 648, 649
member selection operator (.) 610, 655, 793, 908
member-function parameter 566
member-function argument 566
members 383
memberwise assignment 627, 674
memberwise copy 693
memchr 340
memchr function 337, 340
memcmp 337, 339
memcpy 337, 339
memmove function 337, 339
memory 4, 14, 15, 33
memory access violation 262
memory addresses 254
memory allocation 254
memory allocation violation 262
memory consumption 818
memory functions of the string handling library 337
<memory> header file 534, 907
Index
operator overloading 532, 548, 673, 852
decrement operators 698
increment operators 698
Operator Overloads in Templates 849
operator precedence 41
operator precedence chart 919
Operator sizeof when applied to an
array name returns the number of
bytes in the array 271
operator void* member function 878
operator! member function 681, 878
operator!= 695
operator() 720
operator[]
const version 696
non-const version 696
operator+ 674
operator++ 699, 704
operator++( int ) 699
operator<< 680, 692
operator= 694
operator== 695
operator>> 680, 692

operators 77
<< (stream insertion operator) 532
arrow member selection (->) 610
binary scope resolution (::) 586
delete 682
dot (.) 565
member selection (.) 610
new 682
sizeof 609
typeid 825
unary scope resolution (::) 544
operators that can be overloaded 675
optimizations on constants 636
optimizing compiler 163
order 55, 56
order in which constructors and
destructors are called 623
order in which destructors are called 621
order of evaluation
of operators 35
order of evaluation of operands 173
order of exception handlers 918
order of operands of operators 173
original format settings 876
ostream class 854
out-of-range array subscript 897
out-of-range element 692
out_of_range exception 910
outer block 164
output a floating-point value 868
output buffering 879
output data items of built-in type 856
output device 4
output format of floating-point numbers
873
output of char * variables 857
output of characters 857
output of floating-point values 857
output of integers 857
output of standard data types 857
output of uppercase letters 857
output to string in memory 534
output unit 4
oval symbol 57
overflow 518, 897
overflow error 664

overflow_error exception 910
overhead of an extra function call 695
overload an operator as a nonmember,
non-friend function 677
overload the addition operator (+) 674
overload unary operator ! 681
overloaded [] operator 692
overloaded << operator 677
overloaded addition assignment operator
(+=) 700
overloaded assignment (=) operator 691,
694, 698
overloaded binary operators 675
overloaded cast operator function 697
overloaded equality operator (==) 691,
695
overloaded function 833, 836
overloaded function call operator () 698
overloaded increment operator 700
overloaded inequality operator 691, 695
overloaded negation operator 698
overloaded operator += 704
overloaded operator[] member
function 696
overloaded postfix increment operator
700, 704
overloaded prefix increment operator
700, 704
overloaded stream insertion and stream
extraction operators 679
overloaded subscript operator 692, 696
overloaded unary operators 675
overloading 532, 545
<< and >> 548
a member function 609
function definitions 546
operators 548
overloading + 676
overloading += 676
overloading an assignment operator 676
overloading binary operator < 682
overloading binary operators 682
overloading function call operator ()
698, 720
overloading postfix increment operator
699, 704
overloading prefix and postfix decrement
operators 700
overloading prefix and postfix increment
operators 700
overloading resolution 837
overloading stream insertion and stream
extraction operators 678, 691, 692,
700, 704
overloading template functions 837
override a function 792
overtime pay problem 91

P
π 50, 137

%p conversion specifier 362
Package Inheritance Hierarchy 830
Package inheritance hierarchy 776
Package inheritance hierarchy exercise

776
packets in a computer network 472
pad with specified characters 857
padding 404

959

padding characters 865, 868, 869, 871
page layout software 310
palettes in Allegro VI
palindrome 251
palindrome problem 94
parallelogram 729
parameter 144, 566, 568
parameter in the UML 569
parameter list 146, 177, 568, 578
parameter of a function 145
parameter passing 262
parameter types 270
parameterized stream manipulator 854,
863, 866
parameterized type 838, 850
parent node 479
parentheses () 35, 41
parentheses operator (()) 35
partitioning step of Quicksort LXXVI
Pascal programming language 10
Pascal, Blaise 10
pass-by-reference 212, 537, 538
with reference parameters 538
pass-by-value 537
passing an array 213
passing an array element 213
passing an object by value 629
passing arguments by value and by
reference 538
Passing arrays and individual array
elements to functions 213
passing large objects 539
Payroll System Modification 830
Payroll System Modification exercise 830
PDP-11 7
peek function of istream 861
percent sign (%) 34
perfect number 188
perform a task 564
performance 9, 533
performance requirements 163
persistent 5
personal computer 3, 5
Phishing Scanner 452
Pig Latin exercise 349
pipe symbol (|) 508
piping 508
Plauger, P.J. 532
play_fli LII
play_midi LI
play_sample XVII, XX
playing .fli format animations in
Allegro LII
playing MIDI files in Allegro LI
plus sign 870
plus sign, + (UML) 565
Point Class 888
Point class represents an x-y coordinate
pair 783
pointer 254, 255, 258
pointer arithmetic 263, 273, 276, 350
pointer arrow (->) operator 386
Pointer comparisons 275
pointer expression 276
pointer handle 609
pointer manipulation 818
pointer notation 260, 276, 278
pointer parameter 259
pointer subscripting 276


Quadrilateral Inheritance Hierarchy 776
qualified name 761
queue 254, 383, 455, 472, 473, 665
<queue> header file 533
Quick Info box CVIII
Quicksort LXXVI
quicksort 176, LXXVI
quit debugger command CXXIII

R
r file open mode 425
r+ file open mode 425, 426
radians 143
radius 95
raise 518
raising an integer to an integer power 175
RAND_MAX 684
rand 430, 433
raise 153
random number generation 280, 348
random number 152
randomizing 156
range checking 684
Rational Software Corporation 554
RationalNumber class 725
raw array 684
rb binary file open mode 516
rb file open mode 425
rb+ binary open file mode 516
rb+ file open mode 425
rdstate function of ios_base 878
read a line of text 567
read characters with getline 567
read function of ifstream 861
read member function 862
readability 40, 74, 100, 144
readkey IX
real number 7
reassign a reference 541
receiving section of the computer 4
record 265, 419, 421
record key 419
recover from errors 877
rectangle 59
Rectangle class exercise 634
rectangle symbol 57, 64
recursion 167, 174
recursion examples (cont.)
linked list delete 176
linked list insert 176
mazes traversal 175
minimum value in an array 175
multiply two integers 175
postorder traversal of a binary tree 176
preorder traversal of a binary tree 176
print a linked list backwards 176
print a string backwards 175
print an array 175
print an array backwards 175
print a string input at the keyboard backwards 175
printing keyboard inputs in reverse 175
quick sort 176
raising an integer to an integer power 175
recursive main 175
Recursive Selection Sort LXXV
recursive selection sort LXXV
recursive step of Quicksort LXXVI
recursively search a list 492
red breakpoint circle, solid CIV
red input output symbol 408
red input output symbol < 508
red input output symbol > 509
redundant parentheses 38
reference 852
reference parameter 537, 539
reference to a constant 540
reference to a variable 492
reference to a private data member 624
reference to an automatic variable 541
reference to an int 538
reference to an object 606
references must be initialized 541
register 161, 162, 256
reinventing the wheel 8, 141, 532
relational operators 38
release dynamically allocated memory 694
reliable integer division XCVI
remainder 143
remainder operator (%) 34, 51, 153
remove_timer XXXVIII
repetition statement 56, 63
replacement node 493
replacement text 201, 497
requesting a service from an object 562
requires 163, 553
reserved word 42
Resource Centers www.deitel.com/ResourceCenters.html 3
resource leak 901, 903
restore a stream’s state to “good” 878
restrict XCVI
restricted pointer XCVI
resumption model of exception handling 895
rthrow an exception 898
Rethrowing Exceptions 918
return 257
return 0 32
return a result 32
return from a function 142, 143
return key 14, 306
return statement 147, 573
return type 270, 564
void 564, 573
return value type 146, 177
return without expression XCVIII
returning a reference from a function 541
returning a reference to a private data member 624
Returning Error Indicators from Class Time’s set Functions exercise 634
reusability 837, 840
reusable componentry 11
reusable software 9
reuse 553, 579, 609
reuse classes 552
reusing components 12
rewind 517
Richards, Martin 7
Richer Shape Hierarchy 776
right brace (} 25, 26
right child 478
right justification 107, 363
rightmost (trailing) arguments 542
right-shift (>) operator 349, 414
right-shift operator (>>) 855
right-shift operator (>>>) 855
rise-and-shine algorithm 55
Ritchie, D. 7, 15
robust application 890, 896
roll two dice 245
Rolling a six-sided die 6000 times 154
round 167
rounded 72
Rounding 887
rounding 48, 167, 357
rounding toward negative infinity XCVII
rounding toward zero XCVII
rows 229
RTTI (runtime type information) 780, 822, 827
rules of operator 35
Rumbaugh, James 553, 554
run debugger command CXX
runtime error 14, 208
runtime type information (RTTI) 534, 780, 822, 827
runtime_error class 892, 903, 910
what function 897
real (“right value”) 120, 540, 692, 696

S
SalariedEmployee class header file 806
SalariedEmployee class implementation file 807
SalesPerson class definition 612
Sudoku 252, 937, 942, 943
subtracting two pointers 275
subtracting one pointer from another 273
subtract an integer from a pointer 273
substring length 698
substring 698
structure
structure member (.) operator 386, 387, 392
Structure member operator and structure
pointer operator 387
structure pointer (->) operator 386, 387, 392
structure tag name 383, 384
structure type 383
structure variable 385
structured programming 2, 4, 10, 11, 24, 42, 55, 56, 521
structured programming summary 121
structured systems analysis and design 11
Structures 383
Student Inheritance Hierarchy 775
Student poll analysis program 203
student poll program 203
student resources xxvi
sub script 196
subclass 728
subscript 205
subscript notation 266
subscripted name used as an "routine" 692
substr member function of class
string 591
substr member function of string 707
substring 698
substring length 698
subtract an integer from a pointer 273
subtracting one pointer from another 273
subtracting two pointers 273
substraction 4
Sudoku 252, 937, 942, 943
3 by 3 grid 252, 937, 938, 939, 942, 943
9 by 9 grid 252, 937, 938
array 944
beginner’s resources 938
brute force problem solving approach 942, 943
cell 252, 937
column 943
column numbering 252, 937
double 939, 942
exhaustive brute force approach 943
forming puzzles 944
function 943
game players 938
generating a puzzle 943
Sudoku (cont.)
heuristic problem-solving approach 942, 944
hidden double 939
hidden triple 942
history 938
keep your options open heuristic 944
lookahead heuristic for solving
Sudoku 944
mobile device games 938
nested iteration structure 942
player forums 938
programming puzzle creators 252, 937
programming puzzle solvers 252, 937
programming Sudoku puzzle solvers 942
programs 252, 938
puzzle 252, 937
puzzle maker software 938
Resource Center 252, 938, 938, 942
row 943
row numbering 252, 937
simple solution strategies 252, 937
singleton 939, 942
solution strategies 938
solver 938
strategies 939
timer 938
triple 941, 942
tutorials 938
two-dimensional array 252, 937, 942, 943
utility function 943
worksheets 938
Sudoku Resource Center 252, 938, 942
sum 49
sum of numbers 88
sum of the elements of an array 175, 202
sum of two integers 175
superclass 728
supercomputer 3
supermarket simulation 491
survey data analysis 218, 222
Survey data analysis program 219
swapping values LX, LXIV
switch multiple-selection statement 57, 108, 111
logic 799
with break 111
symbol 51, 57
symbol value 925
symbolic constant 110, 201, 496, 497, 501
synchronize operation of an istream and
an ostream 879
synchronous error 897
syntax error 30, 30, 63, 80, 120
T
tab 26, 27, 41, 52, 58, 369, 374
table 229
tabular format 198
tail of a queue 455, 472
tan 143
tangent 143
Target-Heart-Rate Calculator 96, 600
Tax Plan Alternatives 139
telephone number program 349
telephone-number word problem 451
template 833
template definition 549
template function 549, 834
template keyword 549, 834
template parameter 834, 842
template parameter list 549
templates and friends 845
templates and inheritance 845
temporary <double> representation 107
temporary copy 71
temporary file 517
temporary object 697
terminate 14
terminate a program 906
terminate function 898, 901
terminating execution 666
terminating NULL character 207, 208, 361
terminating null character 311, 312, 323
termination housekeeping 620
termination model of exception handling
895
termination phase 69
termination request 518
ternary conditional operator (?:) 675
ternary operator 60, 173
test state bits after an I/O operation 858
Test-Drive: Body Mass Index Calculator
22
Test-Drive: Carbon Footprint Calculator
22
testing error states 877
text analysis 351
text file 516
text manipulation 202
text processing 310
text_height XXVI
text_length XXVI
texipt printf ex XXVI, XXVII
tgmath.h LXXX
The "FairTax" 139
The Twelve Days of Christmas
111
this pointer 654, 655, 663, 676, 695
this pointer used explicitly 654
this pointer used implicitly and
explicitly to access members of an
object 654
Thompson, Ken 7
throw a conditional expression 896
throw an exception 894
throw an int 896
throw exceptions derived from standard
exceptions 911
throw exceptions not derived from
standard exceptions 911
throw keyword 896
throw list 900
throw point 895
throw standard exceptions 911
throw() exception specification 900
Throwing Exceptions from a catch 917
Throwing the Result of a Conditional
Expression 197
TicTacToe Class exercise 634
tie an input stream to an output stream
879
tilde character (~) 620
vertical spacing 60, 100
vertical tab (‘\v’) 312
vi 12
video I/O 854
virtual destructor 826
virtual function 780, 792, 819, 821
virtual function call 821
virtual function call illustrated 820
virtual function table (vtable) 819
virtual memory 904, 906
virtual memory operating systems 11
virtual screen VII
Visual Basic 10
Visual C# programming language 10
Visual C++ xxviii, 10
Visual C++ 2008 xxvii
Visual C++ 2010 xxvii
Visual C++ Express Edition LXXIX
Visual Studio 12
Visual Studio 2005
Quick Info box CVIII
visualizing recursion 175, 191
void * (pointer to void) 275, 338, 456
void keyword 564, 573
volatile information 4
volatile type qualifier 515
volume of a cube 535
stable 819, 822
stable pointer 822
W
w file open mode 425
w+ file open mode 425
w+ file update mode 425
waiting line 665
“walk off” either end of an array 684
Watch debugger command CXXVIII
Watch window (Visual C++ 2005 debugger) CX, CXI
wb binary file open mode 516
wb file open mode 425
wb+ binary file open mode 516
wb+ file open mode 425
wchar_t character type 854
wctype.h LXXX
what virtual function of class exception 892, 897, 904
while repetition statement 63, 63, 68, 74
white space 41, 374, 863
white-space character 58, 60, 858, 859, 860
width implicitly set to 0 865
width member function of class ios_base 865
width of a bit field 403, 406
width setting 865
Windows 508, 518
World Population Growth 139
World Wide Web (WWW) 6
worst-case runtime for an algorithm LIX
wraparound 704
Write 304
write function of ostream 857, 861
writing the word equivalent of a check amount 352
writing to a file 423
X
x 362
Y
yellow arrow in break mode CVIII
Z
0 (zero) flag 368
zeros and ones 418
zeroth element 196
Thank you for purchasing a new copy of *C™ How to Program, Sixth Edition* by P. J. Deitel, H. M. Deitel. The information below provides instruction on how to access the Companion site.

**To access the Companion Website:**


2. From here you can register as a First-Time User or Returning User.

3. Your student access code will be sent to you by CourseSmart. On the registration page, enter your student access code. Do not type the dashes. You can use lower or uppercase letters.

4. Follow the on-screen instructions. If you need help during the online registration process, simply click on Need Help?

5. Once your personal Login Name and Password are confirmed, you can begin viewing the Companion Website.

**To login to the website for the first time after you've registered:**

Follow step 1 to return to the Companion Website. Then, follow the prompts for "Returning Users" to enter your Login Name and Password.

**Note to Instructors:** For access to the Instructor Resource Center, contact your Pearson Representative.

**IMPORTANT:** The access code on this page can only be used once to establish a subscription to the Companion Website for *C™ How to Program, Sixth Edition*. If this access code has already been redeemed, it will no longer be valid. If this is the case, you can purchase a subscription by going to the [http://wps.prenhall.com/ecs_deitel_chtp_6](http://wps.prenhall.com/ecs_deitel_chtp_6) website and selecting "Get Access."