A Beginner's Guide That Makes You Feel SMART

C++ WITHOUT FEAR
SECOND EDITION

void main()
{
    cout << "Hi!";
}

- Learn programming basics fast
- Write your first C++ programs

NEW FEATURES
- Even more figures, examples, and exercises
- Even more puzzles and games
- An expanded 75-page language reference
- Instructions for downloading free C++ software

BRIAN OVERLAND

FREE SAMPLE CHAPTER
SHARE WITH OTHERS
This page intentionally left blank
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data
Overland, Brian R.
C++ without fear : a beginner’s guide that makes you feel smart / Brian Overland.—2nd ed.
p. cm.
Includes index.
1. C++ (Computer program language) I. Title.
QA76.73.C153O838 2011
005.13’3—dc22
2011004218

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-10: 0-13-267326-6
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
Third printing, August 2012
For Colin
This page intentionally left blank
# Contents

**Preface**

- About This Book: How It’s Different xxiii
- Onward to the Second Edition xxiv
- “Where Do I Begin?” xxv
- Icons, Icons, Who’s Got the Icons? xxv
- What Is Not Covered? xxvi
- Getting Started with C++: A Free Compiler xxvii
- A Final Note: Have Fun! xxvii

**Acknowledgments**

xxix

**About the Author**

xxxii

## Chapter 1

**Your First C++ Programs**

- Thinking Like a Programmer 1
  - Computers Do Only What You Tell Them 1
  - Determine What the Program Will Do 1
  - Write the Equivalent C++ Statements 2
  
  **Interlude**  How “Smart” Are Computers, Really? 4
  - Some Nerdy Definitions—A Review 4

- What’s Different About C++? 7

- Building a C++ Program 8
  - Enter the Program Statements 8
  - Build the Program (Compile and Link) 8
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test the Program</td>
<td>9</td>
</tr>
<tr>
<td>Revise as Needed</td>
<td>9</td>
</tr>
<tr>
<td>Installing Your Own C++ Compiler</td>
<td>10</td>
</tr>
<tr>
<td>Example 1.1. Print a Message</td>
<td>11</td>
</tr>
<tr>
<td>If You’re Using the Dev-C++ Environment</td>
<td>12</td>
</tr>
<tr>
<td>If You’re Using Microsoft Visual Studio</td>
<td>12</td>
</tr>
<tr>
<td>How It Works</td>
<td>13</td>
</tr>
<tr>
<td>Exercises</td>
<td>15</td>
</tr>
<tr>
<td>Interlude               What about the <code>#include</code> and <code>using</code>?</td>
<td>15</td>
</tr>
<tr>
<td>Advancing to the Next Print Line</td>
<td>16</td>
</tr>
<tr>
<td>Example 1.2. Print Multiple Lines</td>
<td>16</td>
</tr>
<tr>
<td>How It Works</td>
<td>17</td>
</tr>
<tr>
<td>Exercises</td>
<td>18</td>
</tr>
<tr>
<td>Interlude               What Is a String?</td>
<td>18</td>
</tr>
<tr>
<td>Storing Data: C++ Variables</td>
<td>19</td>
</tr>
<tr>
<td>Introduction to Data Types</td>
<td>20</td>
</tr>
<tr>
<td>Interlude               Why Double Precision, Not Single?</td>
<td>22</td>
</tr>
<tr>
<td>Example 1.3. Convert Temperatures</td>
<td>22</td>
</tr>
<tr>
<td>How It Works</td>
<td>24</td>
</tr>
<tr>
<td>Optimizing the Program</td>
<td>26</td>
</tr>
<tr>
<td>Exercises</td>
<td>28</td>
</tr>
<tr>
<td>A Word about Variable Names and Keywords</td>
<td>28</td>
</tr>
<tr>
<td>Exercise</td>
<td>29</td>
</tr>
<tr>
<td>Chapter 1 Summary</td>
<td>30</td>
</tr>
<tr>
<td>Chapter 2</td>
<td></td>
</tr>
<tr>
<td><strong>Decisions, Decisions</strong></td>
<td>33</td>
</tr>
<tr>
<td>But First, a Few Words about Data Types</td>
<td>33</td>
</tr>
<tr>
<td>Decision Making in Programs</td>
<td>34</td>
</tr>
<tr>
<td>Interlude               What about Artificial Intelligence (AI)?</td>
<td>35</td>
</tr>
<tr>
<td>if and if-else</td>
<td>35</td>
</tr>
<tr>
<td>Interlude               Why Two Operators (= and ==)?</td>
<td>38</td>
</tr>
<tr>
<td>Example 2.1. Odd or Even?</td>
<td>39</td>
</tr>
<tr>
<td>How It Works</td>
<td>40</td>
</tr>
<tr>
<td>Optimizing the Code</td>
<td>42</td>
</tr>
<tr>
<td>Exercise</td>
<td>42</td>
</tr>
<tr>
<td>Introducing Loops</td>
<td>43</td>
</tr>
<tr>
<td>Interlude               Infinite Loopiness</td>
<td>46</td>
</tr>
<tr>
<td>Example 2.2. Print 1 to N</td>
<td>46</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>How It Works</td>
<td>47</td>
</tr>
<tr>
<td>Optimizing the Program</td>
<td>49</td>
</tr>
<tr>
<td>Exercises</td>
<td>49</td>
</tr>
<tr>
<td>True and False in C++</td>
<td>50</td>
</tr>
<tr>
<td><strong>Interlude</strong> The bool Data Type</td>
<td>51</td>
</tr>
<tr>
<td>The Increment Operator (++)</td>
<td>51</td>
</tr>
<tr>
<td>Statements vs. Expressions</td>
<td>52</td>
</tr>
<tr>
<td>Introducing Boolean (Short-Circuit) Logic</td>
<td>53</td>
</tr>
<tr>
<td><strong>Interlude</strong> What Is “true”?</td>
<td>55</td>
</tr>
<tr>
<td>Example 2.3. Testing a Person’s Age</td>
<td>55</td>
</tr>
<tr>
<td>How It Works</td>
<td>56</td>
</tr>
<tr>
<td>Exercise</td>
<td>56</td>
</tr>
<tr>
<td>Introducing the Math Library</td>
<td>57</td>
</tr>
<tr>
<td>Example 2.4. Prime-Number Test</td>
<td>57</td>
</tr>
<tr>
<td>How It Works</td>
<td>59</td>
</tr>
<tr>
<td>Optimizing the Program</td>
<td>60</td>
</tr>
<tr>
<td>Exercise</td>
<td>60</td>
</tr>
<tr>
<td>Example 2.5. The Subtraction Game (NIM)</td>
<td>60</td>
</tr>
<tr>
<td>How It Works</td>
<td>63</td>
</tr>
<tr>
<td>Exercises</td>
<td>63</td>
</tr>
<tr>
<td>Chapter 2 Summary</td>
<td>64</td>
</tr>
</tbody>
</table>

## Chapter 3

**The Handy, All-Purpose “for” Statement**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loops Used for Counting</td>
<td>67</td>
</tr>
<tr>
<td>Introducing the “for” Loop</td>
<td>68</td>
</tr>
<tr>
<td>A Wealth of Examples</td>
<td>70</td>
</tr>
<tr>
<td><strong>Interlude</strong> Does “for” Always Behave Like “while”?</td>
<td>71</td>
</tr>
<tr>
<td>Example 3.1. Printing 1 to N with “for”</td>
<td>72</td>
</tr>
<tr>
<td>How It Works</td>
<td>73</td>
</tr>
<tr>
<td>Exercises</td>
<td>73</td>
</tr>
<tr>
<td>Compound Statements (Blocks) with “for”</td>
<td>74</td>
</tr>
<tr>
<td>Declaring Loop Variables on the Fly</td>
<td>74</td>
</tr>
<tr>
<td>Example 3.2. Prime-Number Test with “for”</td>
<td>75</td>
</tr>
<tr>
<td>How It Works</td>
<td>77</td>
</tr>
<tr>
<td>Exercise</td>
<td>79</td>
</tr>
<tr>
<td>Comparative Languages 101: The Basic “For” Statement</td>
<td>79</td>
</tr>
<tr>
<td>Chapter 3 Summary</td>
<td>80</td>
</tr>
</tbody>
</table>
### Chapter 8  
**Files: Electronic Storage**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introducing File-Stream Objects</td>
<td>197</td>
</tr>
<tr>
<td>How to Refer to Disk Files</td>
<td>199</td>
</tr>
<tr>
<td>Example 8.1. Write Text to a File</td>
<td>200</td>
</tr>
<tr>
<td>How It Works</td>
<td>201</td>
</tr>
<tr>
<td>Exercises</td>
<td>203</td>
</tr>
<tr>
<td>Example 8.2. Display a Text File</td>
<td>203</td>
</tr>
<tr>
<td>How It Works</td>
<td>204</td>
</tr>
<tr>
<td>Exercises</td>
<td>205</td>
</tr>
<tr>
<td>Text Files vs. “Binary” Files</td>
<td>206</td>
</tr>
<tr>
<td><strong>Interlude</strong> Are “Binary Files” Really More Binary?</td>
<td>208</td>
</tr>
<tr>
<td>Introducing Binary Operations</td>
<td>208</td>
</tr>
<tr>
<td>Example 8.3. Random-Access Write</td>
<td>211</td>
</tr>
<tr>
<td>How It Works</td>
<td>213</td>
</tr>
<tr>
<td>Exercises</td>
<td>214</td>
</tr>
<tr>
<td>Example 8.4. Random-Access Read</td>
<td>214</td>
</tr>
<tr>
<td>How It Works</td>
<td>216</td>
</tr>
<tr>
<td>Exercises</td>
<td>217</td>
</tr>
<tr>
<td>Chapter 8 Summary</td>
<td>217</td>
</tr>
</tbody>
</table>

### Chapter 9  
**Some Advanced Programming Techniques**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Arguments</td>
<td>221</td>
</tr>
<tr>
<td>Example 9.1. Display File from Command Line</td>
<td>223</td>
</tr>
<tr>
<td>How It Works</td>
<td>224</td>
</tr>
<tr>
<td>Improving the Program</td>
<td>225</td>
</tr>
<tr>
<td><strong>Interlude</strong> The Virtue of Predefined Constants</td>
<td>226</td>
</tr>
<tr>
<td>Exercises</td>
<td>226</td>
</tr>
<tr>
<td>Function Overloading</td>
<td>227</td>
</tr>
<tr>
<td><strong>Interlude</strong> Overloading and Object Orientation</td>
<td>228</td>
</tr>
<tr>
<td>Example 9.2. Printing Different Types of Arrays</td>
<td>228</td>
</tr>
<tr>
<td>How It Works</td>
<td>230</td>
</tr>
<tr>
<td>Exercise</td>
<td>230</td>
</tr>
<tr>
<td>The do-while Loop</td>
<td>230</td>
</tr>
<tr>
<td>The switch-case Statement</td>
<td>232</td>
</tr>
<tr>
<td>Multiple Modules</td>
<td>234</td>
</tr>
<tr>
<td>Exception Handling: I Take Exception to That!</td>
<td>237</td>
</tr>
<tr>
<td>Say Hello to Exceptions</td>
<td>237</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Private: Members Only (Protecting the Data)</td>
<td>281</td>
</tr>
<tr>
<td>Example 11.1. Testing the Point Class</td>
<td>284</td>
</tr>
<tr>
<td>How It Works</td>
<td>286</td>
</tr>
<tr>
<td>Exercises</td>
<td>286</td>
</tr>
<tr>
<td>Introducing the Fraction Class</td>
<td>286</td>
</tr>
<tr>
<td>Inline Functions</td>
<td>289</td>
</tr>
<tr>
<td>Find the Greatest Common Factor</td>
<td>291</td>
</tr>
<tr>
<td>Find the Lowest Common Denominator</td>
<td>292</td>
</tr>
<tr>
<td>Example 11.2. Fraction Support Functions</td>
<td>293</td>
</tr>
<tr>
<td>How It Works</td>
<td>294</td>
</tr>
<tr>
<td>Exercises</td>
<td>296</td>
</tr>
<tr>
<td>Example 11.3. Testing the Fraction Class</td>
<td>296</td>
</tr>
<tr>
<td>How It Works</td>
<td>299</td>
</tr>
<tr>
<td><strong>Interlude</strong> A New Kind of #include?</td>
<td>299</td>
</tr>
<tr>
<td>Exercise</td>
<td>300</td>
</tr>
<tr>
<td>Example 11.4. Fraction Arithmetic: add and mult</td>
<td>300</td>
</tr>
<tr>
<td>How It Works</td>
<td>304</td>
</tr>
<tr>
<td>Exercises</td>
<td>305</td>
</tr>
<tr>
<td>Chapter 11 Summary</td>
<td>305</td>
</tr>
</tbody>
</table>

**Chapter 12**

**Constructors: If You Build It...**

Introducing Constructors                                            | 307  |
Multiple Constructors (Overloading)                                  | 309  |
C++0x Only: Initializing Members within a Class                       | 309  |
The Default Constructor—and a Warning                                | 310  |
**Interlude** Is C++ Out to Trick You with the Default Constructor?  | 312  |
C++0x Only: Delegating Constructors                                   | 313  |
C++0x Only: Consistent Initialization                                | 314  |

**Example 12.1. Point Class Constructors**                            | 315  |
**How It Works**                                                       | 316  |
**Exercises**                                                           | 317  |

**Example 12.2. Fraction Class Constructors**                          | 317  |
**How It Works**                                                       | 320  |
**Exercises**                                                           | 320  |
Reference Variables and Arguments (&)                                   | 321  |
The Copy Constructor                                                   | 323  |
**Interlude** The Copy Constructor and References                     | 325  |
## Chapter 15  *Two Complete OOP Examples*

- Introducing Linked Lists 389
  - Node Design 390
  - Implementing a Simple Linked List 391
  - An Alphabetical List 393
  - Example 15.1. Names in Alpha Order 395
  - How It Works 397
  - Dealing with Memory Leaks 399
  - C++ Only: Using Smart Pointers to Clean Up 400
  - **Interlude**  Recursion vs. Iteration Compared 401
  - Exercises 402
- Tower of Hanoi, Animated 402
  - Mystack Class Design 403
  - Using the Mystack Class 404
  - Example 15.2. Animated Tower 405
  - How It Works 408
  - Exercises 410
- Chapter 15 Summary 411

## Chapter 16  *Easy Programming with STL*

- Introducing the List Template 413
  - **Interlude**  Writing Templates in C++ 414
<table>
<thead>
<tr>
<th>Chapter 16</th>
<th>Creating and Using a List Class</th>
<th>415</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Creating and Using Iterators</td>
<td>416</td>
</tr>
<tr>
<td></td>
<td>C++0x Only: For Each</td>
<td>418</td>
</tr>
<tr>
<td></td>
<td>Interlude  Pointers vs. Iterators</td>
<td>418</td>
</tr>
<tr>
<td></td>
<td>Example 16.1. STL Ordered List</td>
<td>419</td>
</tr>
<tr>
<td></td>
<td>How It Works</td>
<td>420</td>
</tr>
<tr>
<td></td>
<td>A Continually Sorted List</td>
<td>421</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>422</td>
</tr>
<tr>
<td></td>
<td>Designing an RPN Calculator</td>
<td>422</td>
</tr>
<tr>
<td></td>
<td>Interlude  A Brief History of Polish Notation</td>
<td>424</td>
</tr>
<tr>
<td></td>
<td>Using a Stack for RPN</td>
<td>424</td>
</tr>
<tr>
<td></td>
<td>Introducing the Generalized STL Stack Class</td>
<td>427</td>
</tr>
<tr>
<td></td>
<td>Example 16.2. Reverse Polish Calculator</td>
<td>428</td>
</tr>
<tr>
<td></td>
<td>How It Works</td>
<td>429</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>431</td>
</tr>
<tr>
<td></td>
<td>Correct Interpretation of Angle Brackets</td>
<td>432</td>
</tr>
<tr>
<td></td>
<td>Chapter 16 Summary</td>
<td>432</td>
</tr>
<tr>
<td>Chapter 17</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Inheritance: What a Legacy</strong></td>
<td>435</td>
</tr>
<tr>
<td></td>
<td>How to Subclass</td>
<td>435</td>
</tr>
<tr>
<td></td>
<td>Interlude  Why “public” Base Classes?</td>
<td>437</td>
</tr>
<tr>
<td></td>
<td>Example 17.1. The FloatFraction Class</td>
<td>438</td>
</tr>
<tr>
<td></td>
<td>How It Works</td>
<td>439</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>440</td>
</tr>
<tr>
<td></td>
<td>Problems with the FloatFraction Class</td>
<td>440</td>
</tr>
<tr>
<td></td>
<td>C++ Only: Inheriting Base-Class Constructors</td>
<td>441</td>
</tr>
<tr>
<td></td>
<td>Example 17.2. The Completed FloatFraction Class</td>
<td>442</td>
</tr>
<tr>
<td></td>
<td>How It Works</td>
<td>444</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>445</td>
</tr>
<tr>
<td></td>
<td>Protected Members</td>
<td>445</td>
</tr>
<tr>
<td></td>
<td>Object Containment</td>
<td>447</td>
</tr>
<tr>
<td></td>
<td>Safe Inheritance Through Class Hierarchies</td>
<td>448</td>
</tr>
<tr>
<td></td>
<td>Chapter 17 Summary</td>
<td>451</td>
</tr>
<tr>
<td>Chapter 18</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Polymorphism: Object Independence</strong></td>
<td>453</td>
</tr>
<tr>
<td></td>
<td>A Different Approach to the FloatFraction Class</td>
<td>453</td>
</tr>
<tr>
<td></td>
<td>Virtual Functions to the Rescue!</td>
<td>454</td>
</tr>
</tbody>
</table>
Appendix C  Syntax Summary

Basic Expression Syntax 491
Basic Statement Syntax 492
Control Structures and Branch Statements 493
   The if-else Statement 493
   The while Statement 493
   The do-while Statement 494
   The for Statement 494
   The switch-case Statement 495
   The break Statement 496
   The continue Statement 496
   The goto Statement 497
   The return Statement 497
   The throw Statement 497
Variable Declarations 498
Function Declarations 500
Class Declarations 502
Enum Declarations 503

Appendix D  Preprocessor Directives

The #define Directive 505
The ## Operator (Concatenation) 507
The defined Function 507
The #elif Directive 507
The #endif Directive 508
The #error Directive 508
The #if Directive 508
The #ifdef Directive 509
The #ifndef Directive 510
The #include Directive 510
The #line Directive 511
The #undef Directive 511
Predefined Constants 511
Appendix E  ASCII Codes  513

Appendix F  Standard Library Functions  517
String (C-String) Functions  517
Data-Conversion Functions  517
Single-Character Functions  517
Math Functions  520
Randomization Functions  521
Time Functions  521
Formats for the strftime Function  523

Appendix G  I/O Stream Objects and Classes  525
Console Stream Objects  525
I/O Stream Manipulators  526
Input Stream Functions  528
Output Stream Functions  528
File I/O Functions  529

Appendix H  STL Classes and Objects  531
The STL String Class  531
The <list> Template  533
The <stack> Template  535

Appendix I  Glossary of Terms  537

Index  553
This page intentionally left blank
Many years ago, when I had to learn C overnight to make a living as a programmer (this was before C++), I would have given half my salary to find a mentor, a person who would say, “Here are the potholes in the road...errors that you are sure to make in learning C. And here’s how to steer around them.” Instead, I had to sweat and groan through every error a person could make.

I’m not just talking about programmers who can write or writers who can program. Each of those is rare enough. Much rarer still is the person who is programmer, writer, and teacher—someone who will steer you around the elementary gotchas and enthusiastically communicate the “whys” of the language, including why this stuff is not just useful but, in its own way, kind of cool.

It’s hard to find such a person. But way back then, I swore this is the person I’d become.

Later, at Microsoft, I started in tech support and testing and worked my way into management. But my most important job (I felt) was explaining new technology. I was sometimes the second or third person in the world to see a new feature of a programming language, and my job was to turn a cryptic spec into readable prose for the rest of the universe to understand. I took the goal of “make this simple” as not just a job but a mission.

About This Book: How It’s Different

What’s different about this book is that I’m an advocate for you, the reader. I’m on your side, not that of some committee. I’m aware of all the ways you are “supposed” to program and why they are supposed to be better (and I do discuss those issues), but I’m mostly concerned about telling you what works.

This book assumes you know nothing at all about programming—that you basically know how to turn on a computer and use a mouse. For those of you more knowledgeable, you’ll want to breeze through the first few chapters.
The creators of C and C++—Dennis Ritchie and Bjarne Stroustrup, respectively—are geniuses, and I’m in awe of what they accomplished. But although C and C++ are great languages, there are some features that beginners (and even relatively advanced programmers) never find uses for, at least not for the first few years. I’m not afraid to tell you that information up front: what language features you can and should ignore. At the same time, I’m also eager to tell you about the elegant features of C++ that can save you time and energy.

This is a book about practical examples. It’s also a book about having fun! The great majority of examples in this book either are useful and practical or—by using puzzles and games—are intrinsically entertaining.

So, have no fear! I won’t bludgeon you to death with overly used (and highly abused) terms like data abstraction, which professors love but which forever remain fuzzy to the rest of us. At the same time, there are some terms—object orientation and polymorphism—that you will want to know, and I provide concrete, practical contexts for understanding and using them.

Onward to the Second Edition

The first edition has sold increasingly well over the years. I believe that’s a testament to the variety of learning paths it supplied: complete examples, exercises, and generous use of conceptual art. The second edition builds on these strengths in many ways:

- Coverage of new features in C++0x: This is the new specification for C++ that will be standard by the time you have this book in your hands. Compiler vendors either have brought their versions of C++ up to this standard or are in the process of doing so. This book covers well over a dozen new features from this specification in depth.

- Examples galore, featuring puzzles and games: By the end of Chapter 2, you’ll learn how to enter a program, barely a page long, that not only is a complete game but even has an optimal strategy for the computer. Just see whether you can beat it! But this is only the beginning. This edition features puzzles and games, much more so than the first edition.

- Generous use of conceptual art: The use of clarifying illustrations to address abstract points was one of the biggest strengths of the first edition. This edition has substantially more of these.

- Even more exercises: These encourage the reader to learn in the best way...by taking apart an example that works, analyzing it, and figuring out how to modify it to make it do your own thing.
No-nonsense syntax diagrams: Programming and playing games is fun, but sometimes you need straightforward information. The syntax diagrams in this book, accompanied by loads of examples, clarify exactly how the language works, statement by statement and keyword by keyword.

Introduction to Standard Template Library (STL): Although I lacked the space to do a complete STL manual, this edition (unlike the first) introduces you to the wonders of this exciting feature of C++, showing how it can save you time and energy and enable you to write powerful applications in a remarkably small space.

Expanded reference: The appendixes in the back are intended as a mini desk reference to use in writing C++ programs. This edition has significantly expanded these appendixes.

Essays, or “interludes” for the philosophically inclined: Throughout the book, I detour into areas related to C++ but that impact the larger world, such as computer science, history of programming, mathematics, philosophy, and artificial intelligence. But these essays are set aside as sidebars so as not to interfere with the flow of the subject. You can read them at your leisure.

“Where Do I Begin?”

As I mentioned, this book assumes you know nothing about programming. If you can turn on a computer and use a menu system, keyboard, and mouse, you can begin on page 1. If you already have some familiarity with programming, you’ll want to go through the first two or three chapters quickly.

If you already know a lot about C or C++ and are mainly interested in the new features of C++0x, you may want to go straight to Chapter 10, “New Features of C++0x.”

And if you know C and are now starting to learn about object orientation with the C++ language, you may want to start with Chapter 11, “Introducing Classes: The Fraction Class.”

Icons, Icons, Who’s Got the Icons?

Building on the helpful icons used in the first edition, this edition provides even more—as signposts on the pages to help you find what you need. Be sure to look for these symbols.
These sections take apart program examples and explain, line by line, how and why the examples work. You don’t have to wade through long programming examples. I do that for you! (Or rather, we go through the examples together.)

After each full programming example, I provide at least one exercise, and usually several, that builds on the example in some way. These encourage you to alter and extend the programming code you’ve just seen. This is the best way to learn. The answers can be found on the book’s Web site (www.informit.com/title/9780132673266).

These sections develop an example by showing how it can be improved, made shorter, or made more efficient.

As with “Optimizing,” these sections take the example in new directions, helping you learn by showing how the example can be varied or modified to do other things.

This icon indicates a place where a keyword of the language is introduced and its usage clearly defined.

This icon is used to indicate sections that apply only to versions of C++ compliant with the new C++0x specification. Depending on the version of C++ you have, either these sections will apply to you or they won’t. If your version is not C++0x-compliant, you’ll generally want to skip these sections.

What Is Not Covered?

Relatively little, as it turns out. The two features not covered at all are bit fields and unions. Although these features are useful for some people, their application tends to be highly specialized—limited to a few special situations—and not particularly useful to people first learning the language. Of course, I encourage you to learn about them on your own later.

Another area in which I defer to other books is the topic of writing your own template classes, which I touch on just briefly in Chapter 16. Without a doubt, the ability to write new template classes is one of the most amazing features of state-of-the-art C++, but it is a very advanced and complex topic. For me to cover it adequately and exhaustively could easily have taken another 400 or 500 pages!

And remember that Chapter 16 does introduce you to using STL, which provides extremely useful, existing templates for you to take advantage of.

Getting Started with C++: A Free Compiler

Although this edition doesn’t come with a CD with a free compiler on it, that is no longer necessary. You can download some excellent shareware (that is, free) versions of C++ from the Internet that not only have a free compiler (that’s the application that translates your programs into machine-readable form) but also a very good development environment. And they install easily.

To download this free software, start by going to the book’s Web site: www.informit.com/title/9780132673266.

As mentioned earlier, you will also find downloadable copies of all the full program examples in the book, as well as answers to exercises.

A Final Note: Have Fun!

Once again, there is nothing to fear about C++. Yes, there are those nasty potholes I started out discussing, but remember, I’m going to steer you around them. Admittedly, C++ is not a language for the weak of heart; it assumes you know exactly what you’re doing. But it doesn’t have to be intimidating. I hope you use the practical examples and find the puzzles and games entertaining. This is a book about learning and about taking a road to new knowledge, but more than that, it’s a book about enjoying the ride.
This page intentionally left blank
I am likely to leave many deserving people out this time, but a few names cry out for special mention. The book’s editor, Peter Gordon, not only took the initiative in arranging for the new edition but did a lovely job of nursing the book through all its stages along with the author’s ego. His long-suffering right hand, Kim Boedigheimer, was a better person than we all deserved, coming to the rescue again and again and kindly aiding the author. I’d also like to extend a special thanks to Kim Wimpsett and Anna Popick, who unexpectedly have been an absolute delight to work with in getting the book through its final tense stages.

Closer to home in the Seattle area: I also want to make special mention to veteran Microsoft programmers John R. Bennett and Matt Greig, who provided superb insights about the latest directions of C++. Some of the more interesting new sections in the book came about directly as a result of extended conversations with these experts.
This page intentionally left blank
Brian Overland published his first article in a professional math journal at age 14.

After graduating from Yale, he began working on large commercial projects in C and Basic, including an irrigation-control system used all over the world. He also tutored students in math, computer programming, and writing, as well as lecturing to classes at Microsoft and at the community-college level. On the side, he found an outlet for his lifelong love of writing by publishing film and drama reviews in local newspapers. His qualifications as an author of technical books are nearly unique because they involve so much real programming and teaching experience, as well as writing.

In his 10 years at Microsoft, he was a tester, author, programmer, and manager. As a technical writer, he became an expert on advanced utilities, such as the linker and assembler, and was the “go-to” guy for writing about new technology. His biggest achievement was probably organizing the entire documentation set for Visual Basic 1.0 and having a leading role in teaching the “object-based” way of programming that was so new at the time. He was also a member of the Visual C++ 1.0 team.

Since then, he has been involved with the formation of new start-up companies (sometimes as CEO). He is currently working on a novel.
The most fundamental building block in the programming toolkit is the function—often known as procedure or subroutine in other languages. A function is a group of related statements that accomplish a specific task. Once you define a function, you can execute it whenever you need to do so.

Understanding functions is a crucial step to programming in C++: Without functions, it would be a practical impossibility to engage in serious programming projects. Imagine how difficult it would be to write a word processor, for example, without some means of dividing the labor. Functions make this possible.

The Concept of Function

If you’ve followed the book up until this point, you’ve already seen use of a function—the sqrt function, which takes a single number as input and returns a result.

```cpp
double sqrt_of_n = sqrt(n);
```

This is not far removed from the mathematical concept of function. A function takes zero or more inputs—called arguments—and returns an output, called a return value. Here’s another example. This function takes two inputs and returns their average:

```cpp
cout << avg(1.0, 4.0);
```

Once a function is written, you can call it any number of times. By calling a function, you transfer execution of the program to the function-definition code, which runs until it is finished or until it encounters a return statement; execution then is transferred back to the caller.

This may sound like a foreign language if you’re not used to it. It’s easy to see in a conceptual diagram. In the following example, the program 1) runs normally
until it calls the function avg, passing the arguments a and b, and 2) as a result, the program transfers execution to avg. (The values of a and b are passed to x and y, respectively.)

```c++
void main() {
    double a = 1.2;
    double b = 2.7;
    cout << "Avg is" << avg(a,b);
    cout << endl;
    cout << endl;
    system("PAUSE");
}

double avg(double x, double y) {
    double v = (x + y)/2;
    return v;
}
```

The function runs until it encounters return, at which point: 3) execution returns to the caller of the function, which in this case prints the value that was returned. Then, 4) execution resumes normally inside main, and the program continues until it ends.

```c++
void main() {
    double a = 1.2;
    double b = 2.7;
    cout << "Avg is" << avg(a,b);
    cout << endl;
    cout << endl;
    system("PAUSE");
}

double avg(double x, double y) {
    double v = (x + y)/2;
    return v;
}
```

Note that only main is guaranteed to be executed. Other functions run only as called. But there are many ways a function can be called. For example, main can call a function A, which in turn calls B and C, which in turn calls D.
The Basics of Using Functions

I recommend the following approach for creating and calling user-defined functions:

1. At the beginning of your program, declare the function.
2. Somewhere in your program, define the function.
3. Other functions can then call the function.

**Step 1: Declare (Prototype) the Function**

A function declaration (or prototype) provides type information only. It has this syntax:

```
return_type   function_name (argument_list);
```

The `return_type` is a data type indicating what kind of value the function returns (what it passes back). If the function does not return a value, use `void`.

The `argument_list` is a list of zero or more argument names—separated by commas if there are more than one—each preceded by the corresponding type. (Technically, you don’t need the argument names in a prototype, but it is a good programming practice.) For example, the following statement declares a function named `avg`, which takes two arguments of type `double` and returns a `double` value.

```
double avg(double x, double y);
```

The `argument_list` may be empty, which indicates that it takes no arguments.

**Step 2: Define the Function**

The function definition tells what the function does. It uses this syntax:

```
return_type   function_name (argument_list) {
    statements
}
```

Most of this looks like a declaration. The only thing that’s different is that the semicolon is replaced by zero or more statements between two braces ({}). The braces are required no matter how few statements you have. For example:

```
double avg(double x, double y) {
    return (x + y) / 2;
}
```
The `return` statement causes immediate exit, and it specifies that the function returns the amount \((x + y) / 2\). Functions with no return value can still use the `return` statement but only to exit early.

```cpp
return;
```

**Step 3: Call the Function**

Once a function is declared and defined, it can be used—or rather, *called*—any number of times, from any function. For example:

```cpp
n = avg(9.5, 11.5);
n = avg(5, 25);
n = avg(27, 154.3);
```

A function call is an expression: As long as it returns a value other than `void`, it can be used inside a larger expression. For example:

```cpp
z = x + y + avg(a, b) + 25.3;
```

When the function is called, the values specified in the function call are passed to the function arguments. Here’s how a call to the `avg` function works, with sample values 9.5 and 11.5 as input. These are *passed* to the function, as arguments. When the function returns, the value in this case is assigned to `z`.

```
z = avg(9.5, 11.5);
```

```
double avg(double x, double y) {
    return (x + y) / 2;
}
```

```
(9.5 + 11.5) / 2
21.0 / 2
10.5
```

Another call to the function might pass different values—in this case, 6 and 26. (Because these are integer values, they are implicitly converted, or *promoted*, to type `double`.)
Example 4.1. The avg() Function

This section shows a simple function call in the context of a complete program. It demonstrates all three steps: declare a function, define it, and call it.

```
avg.cpp

#include <iostream>
using namespace std;

// Function must be declared before being used.
double avg(double x, double y);

int main() {

double a = 0.0;
double b = 0.0;

cout << "Enter first number and press ENTER: ";
cin >> a;
cout << "Enter second number and press ENTER: ";
cin >> b;

// Call the function avg().
cout << "Average is: " << avg(a, b) << endl;
```

continued on next page
How It Works

This code is a very simple program, but it demonstrates the three steps I outlined earlier:

1. **Declare** (that is, prototype) the function at the beginning of the program.
2. **Define** the function somewhere in the program.
3. **Call** the function from within another function (in this case, main).

Although function declarations (prototypes) can be placed anywhere in a program, you should almost always place them at the beginning. The general rule is that functions must be declared before being called. (They do not, however, have to be defined before being called, which makes it possible for two functions to call each other.)

```c
double avg(double x, double y);
```

The function definition for the avg function is extremely simple, containing only one statement. In general, though, function definitions can contain as many statements as you want.

```c
double avg(double x, double y) {
    return (x + y)/2;
}
```

The main function calls avg as part of a larger expression. The computed value (in this case, the average of the two inputs, a and b) is returned to this statement in main, which then prints the result.

```c
cout << "Average is: " << avg(a, b) << endl;
```
Function Call a Function!

A program can have any number of functions. For example, you could have two functions in addition to `main`, as in the following version of the program. Lines that are new or changed are in bold.

```cpp
#include <iostream>
using namespace std;

// Functions must be declared before being used.

void print_results(double a, double b);
double avg(double x, double y);

int main() {
    double a = 0.0;
    double b = 0.0;

    cout << "Enter first number and press ENTER: ";
    cin >> a;
    cout << "Enter second number and press ENTER: ";
    cin >> b;

    // Call the function pr_results().
    print_results(a, b);

    system("PAUSE");
    return 0;
}

// print_results function definition
//
void print_results(double a, double b) {
    cout << "Average is: " << avg(a, b) << endl;
}
```

continued on next page
avg2.cpp, cont.

```cpp
// Average-number function definition
double avg(double x, double y) {
    return (x + y)/2;
}
```

This version is a little less efficient, but it illustrates an important principle:
You are not limited to only one or two functions. The program creates a flow of
control as follows:

```
main() → print_results() → avg()
```

EXERCISES

Exercise 4.1.1. Write a program that defines and tests a factorial function. The fac-
torial of a number is the product of all whole numbers from 1 to N. For example,
the factorial of 5 is 1 * 2 * 3 * 4 * 5 = 120. (Hint: Use a `for` loop as described in
Chapter 3.)

Exercise 4.1.2. Write a function named print_out that prints all the whole numbers
from 1 to N. Test the function by placing it in a program that passes a number n
to print_out, where this number is entered from the keyboard. The print_out
function should have type `void`; it does not return a value. The function can be
called with a simple statement:

```
print_out(n);
```

Example 4.2. **Prime-Number Function**

Chapter 2 included an example that was actually useful: determining whether a
specified number was a prime number. We can also write the prime-number test
as a function and call it repeatedly.

The following program uses the prime-number example from Chapters 2 and
3 but places the relevant C++ statements into their own function, is_prime.
// Function must be declared before being used.
bool prime(int n);

int main() {
    int i;

    // Set up an infinite loop; break if user enters 0.
    // Otherwise, evaluate n from prime-ness.
    while (true) {
        cout << "Enter num (0 = exit) and press ENTER: ";
        cin >> i;
        if (i == 0)       // If user entered 0, EXIT
            break;
        if (prime(i))       // Call prime(i)
            cout << i << " is prime" << endl;
        else
            cout << i << " is not prime" << endl;
    }
    system("PAUSE");
    return 0;
}

// Prime-number function. Test divisors from 2 to sqrt of n. Return false if a divisor
// found; otherwise, return true.

bool prime(int n) {
    int i;

    for (i = 2; i <= sqrt(n); i++) {
        if (n % i == 0)       // If i divides n evenly,
            return false;     //  n is not prime.
    }
    return true;       // If no divisor found, n is prime.
}
How It Works

As always, the program adheres to the pattern of 1) declaring function type information at the beginning of the program (prototyping the function), 2) defining the function somewhere in the program, and 3) calling the function.

The prototype says that the prime function takes an integer argument and returns a `bool` value, which will be either `true` or `false`. (Note: If you have a really old compiler, you may have to use the `int` type instead of `bool`.)

```c
bool prime(int n);
```

The function definition is a variation on the prime-number code from Chapter 3, which used a `for` loop. If you compare the code here to Example 3.2 on page 75, you'll see only a few differences.

```c
bool prime(int n) {
    int i;

    for (i = 2; i <= sqrt(n); i++) {
        if (n % i == 0)     // If i divides n evenly,
            return false;         //  n is not prime.
    }
    return true;  // If no divisor found, return
                   // true.
}
```

Another difference is that instead of setting a Boolean variable, is_prime, this version returns a Boolean result. The logic here is as follows:

For all whole numbers from 2 to the square root of n,

If n is evenly divisible by the loop variable (i),

Return the value false immediately.

Remember that the modulus operator (%) carries out division and returns the remainder. If this remainder is 0, that means the second number divides the second evenly—in other words, it is a divisor or factor of the second number.

The action of the `return` statement here is key. This statement returns immediately—causing program execution to exit from the function and passing control back to `main`. There's no need to use `break` to get out of the loop.

The loop in the main function calls the prime function. The use of a `break` statement here provides an exit mechanism, so the loop isn't really infinite. As soon as the user enters 0, the loop terminates and the program ends. Here I've put the exit lines in bold.
Local and Global Variables

while (true) {
    cout << "Enter num (0 = exit) and press ENTER: 
    ";
    cin >> i;
    if (i == 0)           // If user entered 0, EXIT
        break;
    if (prime(i))                   // Call prime(i)
        cout << i << " is prime" << endl;
    else
        cout << i << " is not prime" << endl;
}

The rest of the loop calls the prime function and prints the result of the prime-number test. Note that the prime function, in this case, returns a true/false value, and so the call to prime(i) can be used as an if/else condition.

EXERCISES

Exercise 4.2.1. Optimize the prime-number function by calculating the square root of n only once during each function call. Declare a local variable sqrt_of_n of type double. (Hint: A variable is local if it is declared inside the function.) Then use this variable in the loop condition.

Exercise 4.2.2. Rewrite main so that it tests all the numbers from 2 to 20 and prints out the results, each on a separate line. (Hint: Use a for loop, with i running from 2 to 20.)

Exercise 4.2.3. Write a program that finds the first prime number greater than 1 billion (1,000,000,000).

Exercise 4.2.4. Write a program that lets the user enter any number n and then finds the first prime number larger than n.

Local and Global Variables

Nearly every programming language has a concept of local variable. As long as two functions mind their own data, as it were, they won’t interfere with each other.

That’s definitely a factor in the previous example (Example 4.2). Both main and prime have a local variable named i. If i were not local—that is, if it was shared between functions—then consider what could happen.
First, the `main` function executes `prime` as part of evaluating the `if` condition. Let’s say that `i` has the value 24.

```cpp
if (prime(i))
    cout << i << " is prime" << endl;
else
    cout << i << " is not prime" << endl;
```

The value 24 is passed to the `prime` function.

// Assume i is not declared here, but is global.

```cpp
int prime(int n) {
    for (i = 2; i <= sqrt((double) n); i++)
        if (n % i == 0)
            return false;
    return true;  // If no divisor found, n is prime.
}
```

Look what this function does. It sets `i` to 2 and then tests it for divisibility against the number passed, 24. This test passes—because 2 does divide into 24 evenly—and the function returns. But `i` is now equal to 2 instead of 24.

Upon returning, the program executes

```cpp
cout << i << " is not prime" << endl;
```

which prints the following:

```
2 is not prime
```

This is not what was wanted, since we were testing the number 24!

So, to avoid this problem, declare variables local unless there is a good reason not to do so. If you look back at Example 2.3, you’ll see that `i` is local; `main` and `prime` each declare their own version of `i`.

Is there ever a good reason to not make a variable local? Yes, although if you have a choice, it’s better to go local, because you want functions interfering with each other as little as possible.

You can declare global—that is, nonlocal—variables by declaring them outside of any function definition. It’s usually best to put all global declarations near the beginning of the program, before the first function. A variable is recognized only from the point it is declared, to the end of the file.

For example, you could declare a global variation named `status`:
Recursive Functions

So far, I’ve only shown the use of `main` calling other functions defined in the program, but in fact, any function can call any function. But can a function call itself?

Yes. And as you’ll see, it’s less crazy than it sounds. The technique of a function calling itself is called recursion. The obvious problem is the same one for infinite loops: If a function calls itself, when does it ever stop? The problem is easily solved, however, by putting in some mechanism for stopping.

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int status = 0;

void main () {
    //
}
```

Now, the variable named status may be accessed by any function. Because this variable is global, there is only one copy of it; if one function changes the value of status, this reflects the value of status that other functions see.

**Interlude**

**Why Global Variables at All?**

For reasons shown in the previous section, global variables can be dangerous. Habitual use of global variables can cause shocks to a program, because changes performed by one function cause unexpected effects in another.

But if they are so dangerous, why use them at all?

Well, they are often necessary, or nearly so. Global variables are often the best way to communicate information between functions; otherwise, you might need a long series of argument lists that transfer all the program information back and forth.

Beginning with Chapter 11, we’ll work with classes, which provide an alternative, and generally superior, way for closely related functions to share data with each other: Functions of the same class have access to private data that no one else does.
Remember the factorial function from Exercise 4.1.1 (page 90)? We can rewrite this as a recursive function:

```c
int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);  // RECURSION!
}
```

For any number greater than 1, the factorial function issues a call to itself but with a lower number. Eventually, the function factorial(1) is called, and the cycle stops.

There is a literal stack of calls made to the function, each with a different argument for n, and now they start returning. The stack is a special area of memory maintained by the computer: It is a last-in-first-out (LIFO) mechanism that keeps track of information for all pending function calls. This includes arguments and local variables, if any.

You can picture how to call a factorial(4) this way.

```
factorial(4)
    \downarrow
4 * factorial(3)
    \downarrow
3 * factorial(2)
    \downarrow
2 * factorial(1)
    \downarrow
1
```

Many functions that use a for statement can be rewritten so they use recursion instead. But does it always make sense to use that approach?

No. The example here is not an ideal one, because it causes the program to store all the values 1 through n on the stack, rather than totaling them up directly in a loop. This approach is not efficient. The next section makes a better use of recursion.

Example 4.3. **Prime Factorization**

The prime-number examples we’ve looked at so far are fine, but they have a limitation. They tell you, for example, that a number such as 12,001 is not prime,
but they don’t tell anything more. Wouldn’t it be more useful to know what numbers divide into 12,001?

It’d be more useful to generate the prime factorization for any requested number. This would show us exactly what prime numbers divide into that number. For example, if the number 36 was input, we’d get this:

\[2, 2, 3, 3\]

If 99 was input, we’d get this:

\[3, 3, 11\]

And if a prime number was input, the result would be the number itself. For example, if 17 was input, the output would be 17.

We have almost all the programming code to do this already. Only a few changes need to be made to the prime-number code. To get prime-factorization, first get the lowest divisor, and then factor the remaining quotient. To get all the divisors for a number \(n\), do this:

For all whole numbers from 2 to the square root of \(n\),

- If \(n\) is evenly divisible by the loop variable \(i\),
  - Print \(i\) followed by a comma, and
  - Rerun the function on \(n / i\), and
  - Exit the current function

If no divisors found, print \(n\) itself

This logic is a recursive solution, which we can implement in C++ by having the function get_divisors call itself.

```cpp
#include <iostream>
#include <cmath>
using namespace std;

void get_divisors(int n);

int main() {
  int n = 0;

  cout << "Enter a number and press ENTER: ";
  cin >> n;
```
prime3.cpp, cont.

```cpp
get_divisors(n);
cout << endl;
system("PAUSE");
return 0;
}

// Get divisors function
// This function prints all the divisors of n,
// by finding the lowest divisor, i, and then
// rerunning itself on n/i, the remaining quotient.

void get_divisors(int n) {
    int i;
    double sqrt_of_n = sqrt(n);

    for (i = 2; i <= sqrt_of_n; i++)
        if (n % i == 0) { // If i divides n evenly,
            cout << i << " , " ; // Print i,
            get_divisors(n / i); // Factor n/i,
            return; // and exit.
        }

    // If no divisor is found, then n is prime;
    // Print n and make no further calls.
    cout << n;
}
```

**How It Works**

As always, the program begins by declaring functions—in this case, there is one function other than `main`. The new function is `get_divisors`.

Also, the beginning of the program includes `iostream` and `cmath`, because the program uses `cout`, `cin`, and `sqrt`. You don’t need to declare `sqrt` directly, by the way, because this is done for you in `cmath`.

```cpp
#include <iostream>
#include <cmath>

void get_divisors(int n);
```
The `main` function just gets a number from the user and calls `get_divisors`.

```c
int main() {
    int n = 0;
    cout << "Enter a number and press ENTER: ";
    cin >> n;
    cout << endl;
    system("PAUSE");
    return 0;
}
```

The `get_divisors` function is the interesting part of this program. It has a `void` return value, meaning that it doesn’t pass back a value. But it still uses the `return` statement to exit early.

```c
void get_divisors(int n) {
    int i;
    double sqrt_of_n = sqrt(n);
    for (i = 2; i <= sqrt_of_n; i++)
        if (n % i == 0) {  // If i divides n evenly,
            cout << i <<", ";    // Print i,
            get_divisors(n / i);  // Factor n/i,
            return;               // and exit.
        }
    // If no divisor is found, then n is prime;
    // Print n and make no further calls.
    cout << n;
}
```

The heart of this function is a loop that tests numbers from 2 to the square root of `n` (which has been calculated and placed in the variable `sqrt_of_n`).

```c
for (i = 2; i <= sqrt_of_n; i++)
    if (n % i == 0) {  // If i divides n evenly,
        cout << i <<", ";    // Print i,
        get_divisors(n / i);  // Factor n/i,
        return;               // and exit.
    }
```
If the expression \( n \% i == 0 \) is true, that means the loop variable \( i \) divides evenly into \( n \). In that case, the function does several things: It prints out the loop variable, which is a divisor; calls itself recursively; and exits.

The function calls itself with the value \( n/i \). Because the factor \( i \) is already accounted for, the function needs to get the prime-number divisors for the remaining factors of \( n \), and these are contained in \( n/i \).

If no divisors are found, that means the number being tested is prime. The correct response is to print this number and stop.

```cpp
    cout << n;
```

For example, suppose that 30 is input. The function tests to see what the lowest divisor of 30 is. The function prints the number 2 and then reruns itself on the remaining quotient, 15 (because 30 divided by 2 is 15).

During the next call, the function finds the lowest divisor of 15. This is 3, so it prints 3 and then reruns itself on the remaining quotient, 5 (because 15 divided by 3 is 5).

Here’s a visual summary. Each call to `get_divisors` gets the lowest divisor and then makes another call unless the number being tested is prime.

```
get_divisors(30)
   ↓
print "2," → get_divisors(15)
   ↓
print "3," → get_divisors(5)
   ↓
print "5"
```

### Interlude

A little reflection shows why the lowest divisor is always a prime number. Suppose we test a positive whole number and that \( A \) is the lowest divisor but is not a prime. Since \( A \) is not prime, it must have at least one divisor of its own, \( B \), that is not equal to either 1 or \( A \).

But if \( B \) divides evenly into \( A \) and \( A \) is a divisor of the target number, then \( B \) must also be a divisor of the target number. Furthermore, \( B \) is less than \( A \). Therefore, the hypothesis that the lowest divisor is not prime results in a contradiction.
EXERCISES

Exercise 4.3.1. Rewrite the main function for Example 4.3 so that it prints the prompt message “Enter a number (0 = exit) and press ENTER.” The program should call get_divisors to show the prime factorization and then prompt the user again, until he or she enters 0. (Hint: If you need to, look at the code for Example 4.2, on page 90.)

Exercise 4.3.2. Write a program that calculates triangle numbers by using a recursive function. A triangle number is the sum of all whole numbers from 1 to N, in which N is the number specified. For example, triangle(5) = 5 + 4 + 3 + 2 + 1.

Exercise 4.3.3. Modify Example 4.3 so that it uses a nonrecursive solution. You will end up having to write more code. (Hint: To make the job easier, write two functions: get_all_divisors and get_lowest_divisor. The main function should call get_all_divisors, which in turn has a loop: get_all_divisors calls get_lowest_divisor repeatedly, each time replacing n with n/i, where i is the divisor that was found. If n itself is returned, then the number is prime, and the loop should stop.)

Example 4.4. Euclid's Algorithm for GCF

In the early grades of school, we’re asked to figure out greatest common factors (GCFs). For example, the greatest common factor of 15 and 25 is 5. Your teacher probably lectured you about GCF until you didn’t want to hear about it anymore.

Wouldn’t it be nice to have a computer figure this out for you? We’ll focus just on GCF, because as I’ll show in Chapter 11, if you can figure out the CGF of two numbers, you can easily compute the lowest common multiple (LCM).

The technique was worked out almost 2,500 years ago by a Greek mathematician named Euclid, and it’s one of the most famous in mathematics.

To get CGF: For whole two numbers A and B:

- If B equals 0,
  - The answer is A.
The answer is GCF(B, A%B)

You may remember remainder division (%) from earlier chapters. A%B means this:

Divide A by B and produce the remainder.

For example, 5%2 equals 1, and 4%2 equals 0. A result of 0 means that B divides A evenly.

If B does not equal 0, the algorithm replaces the arguments A, B with the arguments B, A%B and calls itself recursively. This solution works for two reasons:

- The terminal case (B equals 0) is valid. The answer is A.
- The general case is valid: GCF(A, B) equals CGF(B, A%B), so the function calls itself with new arguments B and A%B.

The terminal case, in which B equals 0, is valid assuming A is nonzero. You can see that A divides evenly into both itself and 0, but nothing larger can divide into A. (Note that 0 can be divided evenly by any whole number except itself.) For example, 997 is the greatest common factor for the pair (997, 0). Nothing larger divides evenly into both.

The general case is valid if the following is true:

The greatest common factor of the pair (B, A%B) is also the greatest common factor of the pair (A, B).

It turns out this is true, and because it is, the GCF problem is passed along from the pair (A, B) to the pair (B, A%B). This is the general idea of recursion: Pass the problem along to a simpler case involving smaller numbers.

It can be shown that the pair (B, A%B) involves numbers less than or equal to the pair (A, B). Therefore, during each recursive call, the algorithm uses successively smaller numbers until B is zero.

I save the rest of the proof for an interlude at the end of this section. Here is a complete program for computing greatest common factors:

```
#include <cstdlib>
#include <iostream>
using namespace std;
```
How It Works

All that main does in this case is to prompt for two input variables a and b, call the greatest-common-factor function (gcf), and print results:

```cpp
cout << "GCF = " << gcf(a, b) << endl;
```

As for the gcf function, it implements the algorithm discussed earlier:

```cpp
int gcf(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcf(b, a%b);
}
```

The algorithm keeps assigning the old value of B to A and the value A%B to B. The new arguments are equal or less to the old. They get smaller until B equals 0.
For example, if we start with \( A = 300 \) and \( B = 500 \), the first recursive call switches their order. (This always happens if \( B \) is larger.) From that point onward, each call to \( \text{gcf} \) involves smaller arguments until the terminal case is reached:

<table>
<thead>
<tr>
<th>VALUE OF A</th>
<th>VALUE OF B</th>
<th>VALUE OF A%B (DIVIDE AND GET REMAINDER)</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>500</td>
<td>300</td>
</tr>
<tr>
<td>500</td>
<td>300</td>
<td>200</td>
</tr>
<tr>
<td>300</td>
<td>200</td>
<td>100</td>
</tr>
<tr>
<td>200</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>Terminal case: answer is 100</td>
</tr>
</tbody>
</table>

When \( B \) is 0, the \( \text{gcf} \) function no longer computes \( A\%B \) but instead produces the answer.

If the initial value of \( A \) is larger than \( B \), the algorithm produces an answer even sooner. For example, suppose \( A = 35 \) and \( B = 25 \).

<table>
<thead>
<tr>
<th>VALUE OF A</th>
<th>VALUE OF B</th>
<th>VALUE OF A%B (DIVIDE AND GET REMAINDER)</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>25</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>Terminal case: answer is 5</td>
</tr>
</tbody>
</table>

**Who Was Euclid?**

Who was this Euclid guy? Wasn’t he the Greek who wrote about geometry? (Something like “The shortest distance between two points is a straight line”?)

Indeed he was. Euclid’s *Elements* is one of the most famous books in Western civilization. For almost 2,500 years it was used as a standard textbook in schools. In this work he demonstrated for the first time a *tour de force* of deductive logic, proving all that was then known about geometry. In fact, he invented the whole *idea* of proof. It is a great work that has had profound influence on mathematicians and philosophers ever since.
EXERCISES

Exercise 4.4.1. Revise the program so that it prints out all the steps involved in the algorithm. Here is a sample output:

\[
\text{GCF}(500, 300) => \\
\text{GCF}(300, 200) => \\
\text{GCF}(200, 100) => \\
\text{GCF}(100, 0) => \\
100
\]

Exercise 4.4.2. For experts: Revise the gcf function so that it uses an iterative (loop-based) approach. Each cycle through the loop should stop if \( B \) is zero; otherwise, it should set new values for \( A \) and \( B \) and then continue. You'll need a temporary variable—\( \text{temp} \)—to hold the old value of \( B \) for a couple of lines: \( \text{temp}=b, \) \( b=a\%b, \) and \( a=\text{temp}. \)

Interlude for Math Junkies: Rest of the Proof

Earlier, I worked out some of a proof of Euclid’s algorithm. What remains is to show that the greatest common factor of the pair (\( B, A\%B \)) is also the greatest common factor of the pair (\( A, B \)). This is true if we can show the following:

- If a number is a factor of both \( A \) and \( B \), it is also a factor of \( A\%B \).
- If a number is a factor of both \( B \) and \( A\%B \), it is also a factor of \( A \).
Example 4.5.  **Beautiful Recursion: Tower of Hanoi**

Strictly speaking, the earlier examples don’t require recursion. With some effort, they can be revised as iterative (loop-based) functions. But there is a problem that illustrates recursion beautifully, solving a problem that would be very difficult to solve otherwise.

This is the Tower of Hanoi puzzle: You have three stacks of rings. Each ring is smaller than the one it sits on. The challenge is to move all the rings from the first stack to the third, subject to these constraints:

- You can move only one ring at a time.
- You can place a ring only on top of a larger ring, never a smaller.
It sounds easy, until you try it! Consider a stack four rings high: You start by moving the top ring from the first stack, but where do you move it, and what do you do after that?

To solve the problem, assume we already know how to move a group of N–1 rings. Then, to move N rings from a source stack to a destination stack, do the following:

1. Move N–1 rings from the source stack to the (currently) unused, or “other,” stack.
2. Move a single ring from the source stack to the destination stack.
3. Move N–1 rings from the “other” stack to the destination stack.

This is easier to envision graphically. First, the algorithm moves N–1 rings from the source stack to the “other” stack (“other” being the stack that is neither source nor destination for the current move). In this case, N is 4 and N–1 is 3, but these numbers will vary.

After this recursive move, at least one ring is left at the top of the source stack. This top ring is then moved: This is a simple action, moving one ring from source to destination.

Finally, we perform another recursive move, moving N–1 rings from “other” (the stack that is currently neither source nor destination) to the destination.
What permits us to move \( N-1 \) rings in steps 1 and 3, when the constraints tell us that we can move only one?

Remember the basic idea of recursion. Assume the problem has already been solved for the case \( N-1 \), although this may require many steps. All we have to do is tell the program how to solve the \( N \)th case in terms of the \( N-1 \) case. The program magically does the rest.

It’s important, also, to solve the terminal case, \( N = 1 \). But that’s trivial. Where one ring is involved, we simply move the ring as desired.

The following program shows the C++ code that implements this algorithm:

```cpp
tower.cpp
#include <cstdlib>
#include <iostream>

using namespace std;
void move_rings(int n, int src, int dest, int other);

int main()
{
    int n = 3;  // Stack is 3 rings high
    move_rings(n, 1, 3, 2); // Move stack 1 to stack 3
    system("PAUSE");
}
```
The program is brief considering what it does. In this example, I’ve set the stack size to just three rings, although it can be any positive integer:

```cpp
int n = 3;  // Stack is 3 rings high
```

The call to the move_rings function says that three rings should be moved from stack 1 to stack 3; these are determined by the second and third arguments, respectively. The “other” stack, stack 2, will be used in intermediate steps.

```cpp
move_rings(n, 1, 3, 2); // Move stack 1 to stack 3
```

This small example—moving only three rings—produces the following output. You can verify the correctness of this solution by using three different coins, all of different sizes.

```
Move from 1 to 3
Move from 1 to 2
Move from 3 to 2
Move from 1 to 3
Move from 2 to 1
Move from 2 to 3
Move from 1 to 3
```

Try setting n to 4, and you’ll get a list of moves more than twice as long.
The core of the move_ring function is the following code, which implements the general solution described earlier. Remember, this recursive approach assumes the N–1 case has already been solved. The function therefore passes along most of the problem to the N–1 case.

```cpp
move_rings(n - 1, src, other, dest);
cout << "Move from " << src << " to " << dest
    << endl;
move_rings(n - 1, other, dest, src);
```

Notice how the functional role of the three stacks is continually switched between source (where to move a group of rings from), destination (where the group is going), and other (the intermediate stack, which is not used now but will be at the next level).

**EXERCISES**

**Exercise 4.5.1.** Revise the program so that the user can enter any positive integer value for n. Ideally, you should test the input to see whether it is greater than 0.

**Exercise 4.5.2.** Instead of printing the “Move” message directly on the screen, have the move_ring function call yet another function, which you give the name exec_move. The exec_move function should take a source and destination stack number as its two arguments. Because this is a separate function, you can use as many lines of code as you need to print a message. You can print a more informative message:

```
Move the top ring from stack 1 to stack 3.
```

**Example 4.6.** **Random-Number Generator**

OK, we’ve had enough fun with recursion. It’s time to move on to another, highly practical example. This one generates random numbers—a function at the heart of many game programs.

The test program here simulates any number of dice rolls. It does this by calling a function, rand_0toN1, which takes an argument, n, and randomly returns a number from 0 to n – 1. For example, if the user inputs the number 6, this program simulates dice rolls:

```
3 4 6 2 5 3 1 1 6
```

Here is the program code:
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <ctime>
using namespace std;

int rand_0toN1(int n);

int main() {
    int n, i;
    int r;

    srand(time(NULL)); // Set seed for random numbers.
    cout << "Enter number of dice to roll: ";
    cin >> n;

    for (i = 1; i <= n; i++) {
        r = rand_0toN1(6) + 1; // Get a number 1 to 6
        cout << r << " ";
    }

    system("PAUSE");
    return 0;
}

// Random 0-to-N1 Function.
// Generate a random integer from 0 to N-1, with each
// integer an equal probability.
//
int rand_0toN1(int n) {
    return rand() % n;
}

How It Works

The beginning of the program has to include a number of files to support the
functions needed for random-number generation:
Make sure you include the last three here—cmath, cstdlib, and ctime—whenever you use random-number generation.

Random-number generation is a difficult problem in computing, because computers follow deterministic rules—which, by definition, are nonrandom. The solution is to generate what's called a pseudorandom sequence by taking a number and performing a series of complex transformations on it.

To do this, the program needs a number as random as possible to start off the sequence. So, we're back where we started, aren't we?

Well, fortunately no. You can take the system time and use it as a seed: That is the first number in the sequence.

```c
srand(time(NULL));
```

NULL is a predefined value that means a data address set to nothing. You don’t need to worry about it for now. The effect in this case is simply to get the current time.

The C++0x specification provides the `nullptr` keyword, which should be used in preference to NULL if you have a C++0x-compliant compiler.

A program that uses random numbers should call `srand` first. System time changes too quickly for a human to guess its exact value, and even a tiny difference in this number causes big changes in the resulting sequence. This is a practical application of what chaos theorists call the Butterfly Effect.

The rest of `main` prompts for a number and then prints the quantity of random numbers requested. A `for` loop makes repeated calls to `rand_0toN1`, a function that returns a random number from 0 to n – 1:

```c
for (i = 1; i <= n; i++) {
    r = rand_0toN1(6) + 1;  // Get num from 1 to 6
    cout << r << " ";       // Print it out
}
```

Here is the function definition for the `rand_0toN1` function:

```c
int rand_0toN1(int n) {
    return rand() % n;
}
```
This is one of the simplest functions we’ve seen yet! Calling \texttt{rand} produces a number anywhere in the range of the \texttt{int} type, which, on 32-bit systems, can be anywhere in the range of roughly plus or minus two billion. But we want much smaller numbers.

The solution is to use your old friend, the remainder-division operator (\%), to divide by \( n \) and return the remainder. No matter how large the amount being divided, the result must be a number from 0 to \( n-1 \), which is exactly what the function is being asked to provide.

In this case, the function is called with the argument 6, so it returns a value from 0 to 5. Adding 1 to the number gives a random value in the range 1 to 6, which is what we want.

\textbf{EXERCISES}

\textbf{Exercise 4.4.1.} Write a random-number generator that returns a number from 1 to \( N \) (rather than 0 to \( N-1 \)), where \( N \) is the integer argument passed to it.

\textbf{Exercise 4.4.2.} Write a random-number generator that returns a random floating-point number between 0.0 and 1.0. (Hint: Call \texttt{rand}, cast the result \( r \) to type \texttt{double} by using \texttt{static_cast<double>(r)}, and then divide by the highest value in the \texttt{int} range, \texttt{RAND_MAX}.) Make sure you declare the function with the \texttt{double} return type.

\textbf{Games and More Games}

Now that we know how to write functions and generate random numbers, it’s possible to enhance some game programs.

The Subtraction Game example at the end of Chapter 2 can be improved. Right now, when the user plays optimal strategy, the computer responds by choosing 1. We can make this more interesting by randomizing the computer’s response in these situations. The following program makes the necessary changes, putting altered lines in bold:

```cpp
#include <iostream>
#include <cmath>
#include <ctime>
#include <cstdlib>
```

continues on next page
using namespace std;

int rand_0toN1(int n);

int main() {
    int total, n;

    srand(time(NULL)); // Set seed for random numbers.
    cout << "Welcome to NIM. Pick a starting total: ";
    cin >> total;
    while (true) {

        // Pick best response and print results.
        if ((total % 3) == 2) {
            total = total - 2;
            cout << "I am subtracting 2." << endl;
        } else if ((total % 3) == 1) {
            total--;
            cout << "I am subtracting 1." << endl;
        } else {
            n = 1 + rand_0toN1(2); // n = 1 or 2.
            total = total - n;
            cout << "I am subtracting ";
            cout << n << "." << endl;
        }
        cout << "New total is " << total << endl;
        if (total == 0) {
            cout << "I win!" << endl;
            break;
        }

        // Get user’s response; must be 1 or 2.
        cout << "Enter num to subtract (1 or 2): ";
        cin >> n;
        while (n < 1 || n > 2) {
            cout << "Input must be 1 or 2." << endl;
            cout << "Re-enter: ";
            cin >> n;
        }
    }
}
Chapter 2 presented an exercise: Alter this program so that it permits any number from 1 to N to be subtracted each time, where N is set at the beginning. That problem is left as an exercise for this version as well. (You can even prompt the end user for this value before the game starts. As always, the computer should win whenever the user does not play perfect strategy.)

The last full example in Chapter 10 presents a game of Rock, Paper, Scissors that can be programmed even with C++ compilers that are not fully C++0x compliant. To use the example in Chapter 10 (Example 10.3) with weak, rather than strong, enumerations, replace this line in Chapter 10:

\[
\text{enum class Choice} \{ \text{rock, paper, scissors} \};
\]

with this:

\[
\text{enum Choice} \{ \text{rock, paper, scissors} \};
\]

Also, remove the using statement:

\[
\text{using namespace Choice;}
\]

Chapter 4  **Summary**

Here are the main points of Chapter 4:

- In C++, you can use functions to define a specific task, just as you might use a subroutine or procedure in another language. C++ uses the name function for all such routines, whether they return a value or not.
You need to declare all your functions (other than `main`) at the beginning of the program so that C++ has the type information required. Function declarations, also called *prototypes*, use this syntax:

```cpp
type  function_name (argument_list);
```

You also need to define the function somewhere in the program, to tell what the function does. Function definitions use this syntax:

```cpp
type  function_name (argument_list) {
    statements
}
```

A function runs until it ends or until the `return` statement is executed. A `return` statement that passes a value back to the caller has this form:

```cpp
return  expression;
```

A return statement can also be used in a `void` function (function with no return value) just to exit early, in which case it has a simpler form:

```cpp
return;
```

Local variables are declared inside a function definition; global variables are declared outside all function definitions, preferably before `main`. If a variable is local, it is not shared with other functions; two functions can each have a variable named `i` (for example) without interfering with each other.

Global variables enable functions to share common data, but such sharing provides the possibility of one function interfering with another. It's a good policy not to make a variable global unless there's a clear need to do so.

The addition-assignment operator (`+=`) provides a concise way to add a value to a variable. For example:

```cpp
n += 50;                // n = n + 50
```

C++ functions can use recursion—meaning they call themselves. (A variation on this is when two or more functions call each other.) This technique is valid as long as there is a case that terminates the calls. For example:

```cpp
int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);  // RECURSION!
}
Symbols

-> (access class member) operator, 391, 476
+= (add and assign) operator, 477
- (change sign of) operator, 333, 476
-- (decrement) operator
   associativity, precedence, and syntax of, 476
   safe use of, 55
   using with loops, 52, 70
" " (double quotes)
   distinguishing strings from individual
   characters, 185
   in #include syntax, 510
>= (greater than or equal to) operator. See
   Greater than or equal to (>=) operator
!= (inequality) operator, 47, 477
, (join) operator
   associativity, precedence, and syntax of, 477
   as delimiters in text, 187
   uses of, 482
<<= (left shift and assign) operator, 477
<= (less than or equal to) operator, 47, 477
*= (multiply and assign) operator, 477
.* (pointer-to-member) operator, 476
->* (pointer-to-member) operator, 476
>>= (right shift and assign) operator, 477
-= (subtract and assign) operator, 477
- (subtraction) operator, 342–343, 476
./ (access class member) operator, 476
/= (divide and assign) operator, 477
/ (divide) operator, 476
// (double slashes), in comment syntax, 23–24
:: (scope) operator, 476, 478
; (semicolon)
   class or data declaration ending with, 280
   compound statements and, 36
   termination of function prototypes, 119
   termination of statements, 52–53
   use in program syntax, 14
\ (backslash), 177
| (bitwise OR) operator, 477, 480
|| (logical OR) operator, 477
+ (addition) operator. See Addition (+) operator
++ (increment) operator. See Increment (++) operator
= (assignment) operator. See Assignment operator (=)
== (equality) operator. See Equality operator (==)
& (address) operator, 160, 321, 476
&= (bitwise AND and assign) operator, 477
! (logical negation) operator
   associativity, precedence, and syntax of, 476
   swap function and, 161
   types of Boolean operators, 54
&& (bitwise AND) operator. See AND (&&) operator
## (concatenation) operator, 507
%= (modular divide and assign) operator, 477
% (modulus or remainder) operator
  associativity, precedence, and syntax of,
  declaring, 335–336
  prime number function and, 92
  random number generation and, 113
  using in Odd-or-Even program, 41–42
( ) (function call) operator, 476
* (indirection operator), 149, 166, 476
* (multiply) operator, 476
?: (conditional) operator, 481, 506
[ ] (access array element) operator, 476
^= (bitwise XOR and assign) operator, 477
^ (bitwise XOR) operator, 477, 480
{} (braces). See Braces ({})
~ (bitwise negation) operator, 476, 481
<> (angle brackets)
  correct interpretation of, 432
#include syntax, 299–300, 510–511
<< (bitwise left shift or stream op) operator, 476, 481
< (less than) operator. See Less than (<) operator
>> (bitwise right shift or stream op) operator, 476. 481
> (greater than) operator, 47, 477
>> (stream input) operator, 180

Numbers
16-bit integers (short), 244–245
2-D arrays, 142–143
32-bit
  addresses, 148
  integers (long), 244–245
64-bit
  addresses, 148

Fibonacci numbers as 64-bit example, 250–254
integers (long long), 244–245
literals (constants), in C++Ox specification, 246–247

A
Abstract classes
  declaring an abstract Printable class, 465
  defined, 537
  as a pattern for subclasses, 463
  specifying and enforcing a set of services
    (as an interface), 463–464
  stream classes demonstrating
    extensibility of OOP, 464–466
  virtual functions in, 462
Add and assign (+=) operator, 477
Add function
  adding arithmetic functions to Fraction class, 300–305
  refining in Fraction class, 347–348
Addition (+) operator
  adding to Point class, 342–343
  associativity, precedence, and syntax of,
  declaring, 334
  overloading in Fraction class, 348–349
  using with references, 338–339
Address operator (&), 160, 476
Addresses
  32-bit and 64-bit, 148
  arr constant, 161–163
  comparing address expressions to each other, 163
CPUs determining location by, 145
defined, 537
doubling variable whose address is
passed via pointer, 152–155
pointers and, 146
printing, 151–152
Advanced programming. See Programming,
advanced
Aggregates, for initializing arrays, 119
AI (Artificial intelligence), 35
Algorithms, in data evaluation, 4
Alphabetical lists, 393–395
American National Standards Institute (ANSI)
data types supported by, 483
defined, 537
AND (&&) operator
associativity, precedence, and syntax of,
477
operating on integers of same width, 480
testing a person’s age (example of
Boolean logic), 55–56
types of Boolean operators, 54
Angle brackets (< >)
correct interpretation of, 432
#include syntax, 299–300, 510–511
ANSI (American National Standards Institute)
data types supported by, 483
defined, 537
Applications, defined, 5, 537
argv, command-line argument, 222, 224–225
Arguments
catch statements, 239
command-line, 221–222
compared with pointers, 321
declaring functions and, 83, 85, 500
defined, 538
operators, 335
pointers, 172
references, 322–323
argc, command-line argument, 222
Arithmetic operators
pointers and, 162
precedence of, 54
arr constant, in addressing, 161–163
Arrays
2-D, 142–143
allocating array of any size, 368–370
boundary problems in indexing, 141
Card Dealer #1 (example), 129–132
Card Dealer #2 (example), 132–136
Card Dealer #3 (example), 136–140
as container for range-based for, 257
distinguishing strings from individual
characters, 185–186
initializing, 119
linked lists compared with, 389
overview of, 117
pointers used in processing, 163–164
print_array function (example), 228–230
printing out elements (example),
121–123
range-based "for" (for each) in setting,
258–261
sorting with swap function, 156–161
of strings, 128–129
strings as arrays based on char type, 171
summary, 143–144
testing randomness with, 123–127
uses of, 117–119
zero-based indexes and, 119–120
Artificial intelligence (AI), 35
ASCII code
char type and, 171–172
corresponding to characters, 169
defined, 538
distinguishing strings from individual
characters, 185–186
extended codes for DEC, HEX, and
CHAR, 515
reading string input, 178
special meanings of nonprintable
characters, 513
ASCII code (continued)
    standard codes for DEC, HEX, and CHAR, 514
    strings and, 19
text files vs. "binary" files, 206
Assignment operator (=)
    copy constructors compared with, 380
equality operator (==) compared with, 36, 38–39
    in expression syntax, 491–492
overview of, 349–350
precedence of, 166
return *this statement as last statement in, 379–380
    in statements, 25–26
Associativity
    defined, 538
    in evaluation of expressions, 166–167
    of operators, 475–477
    atof function, 183
    atoi function, 183, 206
    atol() function, 247–248
    auto keyword
        uses of, 261–262
        variable modifiers, 499
        working with exotic types and, 243
    Avg(), 87–88

B
Backslash (\), indicating special meaning, 177
Backward compatibility
    with C language, 281
    defined, 538
    issues with C, 312–313
Base 10 numbers. See Decimal notation (base 10)
Base 16 numbers. See Hexadecimal notation (base 16)
Base 2 numbers. See Binary notation (base 2)
Base 8 (Octal notation), 485
Base classes
    access levels, 437–438
    constructors (C++Ox only), 441–442
    defined, 538
    inheriting from, 435
    principle of passing something specific (subclass) to something more general (base class), 466
    subclasses not inheriting constructors from, 440
    base(), output stream function, 528
    base_check keyword, in explicit overrides (C++Ox only), 460–461
Basic language
    adding OOP extensions to, 279
    purpose of high-level languages, 7
    "for" statement in, 79
Bauer, F. L., 424
BEL (bell), 513–514
Binary files
    operations on, 208–211
    reading data from, 214–217
text files vs., 206–208
    writing data to, 211–214
Binary notation (base 2)
    Fibonacci and, 255
    floating-point data type and, 33
    hexadecimal and decimal equivalents, 147
Bits, 538
Bitwise left shift or stream op (<<) operator, 476
Bitwise negation (~) operator, 476, 481
Bitwise operators
    associativity, precedence, and syntax of, 476
    logical operators compared with, 55
    operating on integers, 480
    uses of, 481
Bitwise right shift or stream op (>>) operator, 476, 481
Bitwise XOR and assign (^=) operator, 477
Bitwise XOR (^) operator, 477, 480
Blocks. See Compound statements (blocks)
Bonacci, Leonardo (Fibonacci), 255

**bool** data type
- array of bool items, 138–139
- C++ support for, 51
- converting logical operators to, 55
- prime number function and, 92–93

**boolalpha**, 527
Boolean algebra, 53
Boolean values (true/false)
- C++ support for **bool** data type, 50–51
- defined, 539
- introducing short-circuit logic, 53–55
- relational operators returning true or false values, 47
- testing a person’s age (example of Boolean logic), 55–56

Boundary checking, arrays and, 141
Braces ({}), 309–310
in **if-else** statements, 37–38
in initialization, 314–315
in program syntax, 13
Branch statements. See also Control structures, 493
**break** statements
- interrupting loops, 46, 60
- syntax of control structures, 496
- transferring control out of loop or function, 231

BS (backspace), 513–514
Bubble sort algorithm, 156–157
Bytes
- defined, 539
- one byte per character in ASCII code, 170

C

C language
- .h files, 511

backward compatibility with, 281, 312–313
C++ built on, 279
purpose of high-level languages, 7
C++, brief history of, 7, 279
C++ compilers, See Compilers
C++Ox specification
- 64-bit literals (constants), 246–247
- **auto** keyword, 261–262
- base class constructors, 441–442
- challenges of **long long** type, 247–248
- consistent initialization, 314–315
- data types supported by, 483
- **decltype** keyword, 262
- defined, 539
- delegating constructors, 313–314
- **enum** classes, 265–266
- extended enum syntax, 266–267
- Fibonacci numbers as 64-bit example, 250–254
- formatting **long long** numbers, 248–249
- initializing members within a class, 309–310
- localizing numbers, 254–255
- new version of **for** loops, 69–70
- **nullptr** keyword, 112, 187, 262–263
- overview of features in, 243–244
- range-based "for" (for each), 256–261
- raw string literals, 273
- Rock, Paper, Scissors game, 267–272
- strongly typed enumerations, 263–265
- summary, 273–275
- supporting **long long int**, 34, 244–246
- user-defined literals, 357–359

C-strings. See Strings (C-strings)
Callback, 539
Calling functions
- arguments in, 227–228
- **avg()** example, 87–88
- as expression, 86–87
- implementation of, 325
Calling functions (continued)
object member functions, 179
overview of, 83
prototypes and, 235
Card dealer array examples
Card Dealer #1, 129–132
Card Dealer #2, 132–136
Card Dealer #3, 136–140
Case
case-sensitivity in compiling code, 11
Convert to Uppercase program
(example), 183–185
switch case statements, 232–234
Casts
associativity, precedence, and syntax of
cast operator, 476
defined, 539
new vs. old style for, 479
catch keyword, in exception handling,
238–239
cctype, 185
Celsius, converting to/from Fahrenheit, 22–26
Central processing units. See CPUs (central
processing units)
cerr, 526
Change sign of (-) operator, 333, 476
char*
converting to long long integers, 247–248
format of string literals, 273, 486
for string variables, 128–129
char type. See also Strings (C-strings)
8 bit, 245
ASCII codes for, 514–515
description and range of, 484
new string class and, 363
strings based on, 171–172
syntax of, 186
Characters
accessing individual characters in new
string type, 193
conversion and testing functions in
standard library, 519
distinguishing strings from individual
characters, 185–186
manipulating individual characters in a
string, 183–184
cin
adding values to variables, 19–20
description of stream objects, 526
file streams as alternative to, 197
getline, 179–180
string type and, 191
class keyword, 279
Classes
abstract. See Abstract classes
arithmetic functions added to, 300–305
constructors for, 317–320
copy constructors for, 324–329
declaring, 279–280, 502–503
defined, 540
destructors, 370–371
encapsulation and, 236
enum class, 265–266
exception class, 240
finding GCF of, 291–292
finding LCM of, 292
Fraction class example. See Fraction class
hierarchies for safe inheritance, 448–451
inheritance. See Inheritance
initializing members within (C++Ox
 only), 309–310
inline functions of, 289–291
lists. See List classes
naming, 266
objects in, 277–278
operators for Fraction class, 352–357
operators for Point class, 340–348
operators for (Print function), 351–352
Point class example. See Point class
polymorphism. See Polymorphism
principle of passing something specific (subclass) to something more general (base class), 466
private members of, 281–284
range-based for contained in, 257
relationship with objects and members, 284
stack class, 425, 427–428
streams. See Streams
strings. See String class
structures and, 281
subclasses. See Subclasses
summary, 305–306
support functions of, 293–296
syntax of class keyword, 279
testing, 284–286, 296–300
clog, 526
close()
closing files, 199
description of file I/O functions, 529
cmath library
accessing, 57
functions in, 520
including, 181
Code
defined, 5, 540
packaging with data in classes, 277–278
as program instructions, 1
Code reuse
inheritance and, 435
object containment and, 447–448
Command-line
arguments, 221–222
displaying files from, 223–226
Commas (,). See Join (,) operator
Comments, use in programs, 23–24
Compilers
as applications, 5
building a C++ program, 8–9
defined, 5, 540
function of, 3
ignoring comments, 24
installing, 10
modules and, 235
overloading and, 228
translating statements into machine code, 170–171
Component models, in systems approach to programming, 472
Compound statements (blocks)
defined, 540
for executing a series of things, 36
if-else used with, 37–38
for loops used with, 74
replacing control structures with, 231
syntax of, 492
while loops used with, 43–45
Computers
ability to make judgments, 35
doing only what you tell them, 1
storing text on, 169–170
Concatenation
defined, 551
strcat and sstrncat functions for, 172–174
working with string type and, 193
writing function for string class, 380–382
Conditional (?:) operator, 477
Conditions
in for loops, 68–69
in while loops, 43–45
Console input, see cin
Console stream objects. See Streams
const keyword
in declaring copy constructors, 324
function modifiers, 501
preventing changes to arguments while passing, 340
variable modifiers, 499
Constants
64-bit literals, 246–247
advantages of predefined, 225–226
all literals are constants, 485
Constanst (continued)
automating assignment of symbolic, 264–265
defined, 540
end line constant, 16
list of predefined, 512
Constructors
for base classes (C++0x only), 441–442
cannot be virtual, 455
consistent initialization of (C++0x only), 314–315
copy constructors, 323–325
copy constructors for Fraction class, 325–329
default constructors, 310–313
defined, 540
delegating (C++0x only), 313–314, 452
for Fraction class, 317–320
initializing members within a class (C++0x only), 309–310
initializing objects from strings, 329–331
for linked lists, 390–391
multiple (overloading), 309
overview of, 307–309
for Point class, 315–317
reference variables and arguments and, 321–323
for String class, 374
subclasses not inheriting, 440
summary, 331–332
Containers
range-based for (for each), 257
templates creating, 413
continue statements, 496
Control structures
defined, 540
do while, 231–232
else. See else
for. See for
if. See if statements
switch case statements, 232–234
syntax of, 493
types of, 230–231
while. See while
Convert to Uppercase program (example), 183–185
Cooked literals, 359
Copy constructors
assignment operator compared with, 380
deep copying, 377
defined, 541
for Fraction class, 325–329
member functions automatically supplied by compiler, 349
overview of, 323
references and, 325
shallow copying, 376–377
syntax of, 324
Counting, loops used for, 67–68
cout
console output object, 14
derscription of stream objects, 526
file streams as alternative to, 197
polymorphism and extensibility of, 464–466
print function interacting with, 351–352
string type and, 191
use of data objects in C++, 7–8
CPUs (central processing units)
addresses of locations, 145
defined, 541
translating statements into machine code prior to execution, 170–171
CR (carriage return), 513–514
cstdlib, 181
ctemp variable, for holding Celsius values, 19–20, 24–25
D
Data
arrays, 117
conversion functions in standard library, 518
defined, 5
linking data structures using addresses, 146
packaging with code in classes, 277–278
pointers for sending large amounts of, 146
programs and, 1
storing via variables, 19–20
string data, 19
Data declaration
classes, 502–503
enumerations, 503–504
functions, 500–501
semicolon terminating, 280
variables, 498–500
Data members. See Members
Data types
classes creating, 277
comparing integers with floating point data, 33–34
description and range of, 484
initializing while declaring, 307
introduction to, 20–22
of numeric literals, 485–486
operations on, 333
polymorphism and, 228
precision of, 484
dec, 527
Decimal notation (base 10)
ASCII codes, 514–515
Fibonacci and, 255
hexadecimal and binary equivalents, 147
numeric format of literals, 485
Decision making
Boolean values (true/false), 50–51
if and if-else, 35–38
increment operator (++) and, 51–52
loops in, 43–46
odd or even, 39–41
optimizing Odd-or-Even program, 42
overview of, 33
print 1 to N loop example, 46–49
in programs, 34–35
short-circuit logic, 53–55
Subtraction Game example (NIM), 60–63
summary, 64–65
testing a person's age, 55–56
testing for prime numbers, 57–60
Declaration
of classes, 502–503
of classes (Point class example), 279–280
defined, 541
of enumerations, 503–504
of operators, 334–336
of pointers, 148–150
Declaration, of functions
overview of, 500–501
termination of function prototypes, 119
user-defined, 85
Declaration, of variables
assigning values while initializing, 49
avg() example, 87–88
on the fly with for loops, 74–75
local and global variables and, 93–95
overview of, 498–500
prior to use, 20
for string type, 189–190
dctype keyword, 243, 262
Decrement operator (--) associativity, precedence, and syntax of, 476
safe use of, 55
using with loops, 52, 70
Deep copying, 377
Default constructors
defined, 541
member functions automatically supplied by compiler, 349
supplying constructors vs. accepting default, 310–313
#define directive
localizing numbers, 254–255
overview of, 505–506
placing predefined constants with, 225–226

defined function, preprocessor directives, 507
Defining functions
avg() example, 87–88
defined, 541
overview of, 85–86
DEL (delete), 513–514
Delegating constructors (C++Ox only), 313–314, 452
delete operator
associativity, precedence, and syntax of, 476
class destructors and, 370–371
pointers to objects and, 365–366
releasing allocated memory, 364–365, 367, 370

Delimiter, in text, 186
Deprecate
defined, 541
setting null pointers without nullptr keyword, 263

Dereference, 542
Derived classes. See also Subclasses, 279, 541

Destructors
class destructors, 370–371
defined, 542
in String class, 374
virtual, 455

Dev-C++, 12
Dijkstra, E. W., 424
Directives. See Preprocessor directives
Directories, referencing disk files in, 199–200
Displaying text files, 203–206
Divisors, lowest divisor as prime number, 100–101
do while statements
as loop, 230

statement and conditions, 232
syntax of, 231, 494

Double-precision types. See double
Double quotes (" ")
distinguishing strings from individual characters, 185
in #include syntax, 510

double type
arrays and, 117–118
comparing integers with floating point data, 33–34
converting strings to, 183
"cooked" literals and, 359
description and range of, 484
as floating-point data type, 21–22
precision of, 484
range compared with int, 127
reading binary data, 209–210
storing literals in, 485
using with ctemp and ftemp variables, 24–25

Double_it function, doubling variable with, 152–155

Dynamic memory allocation
allocating memory blocks with, 366–367
for array of any size, 368–370
new keyword and, 363–364
problems with, 368

E

#define directive, 507–508
else statements
as control structure, 231
in if-else statements, 37–38
syntax of, 493

Encapsulation
defined, 542
fout object and, 198
private/public distinction and, 236

End users. See Users

#define directive, 508
endl stream manipulator
description of, 527
for end line constant, 16
ends stream manipulator, 527
Enumerations
automating assignment of symbolic
constants, 264–265
enum classes, 265–266
enum declarations, 503–504
extended enum syntax, 266–267
Rock, Paper, Scissors game, 267–272
strongly typed, 244, 263–265
eof (end of file) function, file I/O functions,
205, 529
Equality operator (==)
associativity, precedence, and syntax of,
476
compared with assignment operator (=),
36, 38–39
in Fraction class, 355–356
ordered lists and, 419
overview of, 350–351
precedence of, 166
in String class, 374
types of relational operators, 47
#error directive, 508
Errors. See also Exception handling
syntax errors and program-logic errors,
9–10
types of, 237
Escape sequences, strings, 177–178, 486–487
Euclid, 101–105
exception class, 240
Exception handling
examples of exceptions, 237
exceptions as runtime errors, 237
exceptions defined, 542
in small program, 238
throw statements and, 497
try-catch-throw approach to, 238–240
Expressions
evaluating in RPN, 422–423
function calls as, 86–87
precedence and associativity in
evaluating, 166–167
single quote vs. double quotes in syntax
of, 186
statements compared with, 52–53
syntax of, 491–492
Extensibility, of OOP, 464–466
extern declaration
sharing variables and, 235
for variable modifiers, 499–500
F
F suffix, representing float format, 486
Factorial function
overview of, 90
rewriting as recursive function, 96
Factorization, of prime numbers, 96–100
Fahrenheit, converting to/from Celsius, 22–26
False/true. See Boolean values (true/false)
FF (form feed), 513–514
Fibonacci numbers, 250–254
File-error exceptions, 240
File I/O functions, 529–530
File mode flags, 530
File stream objects
associating with disk files, 198
including, 197
Filenames
entering from command-line, 221
prompts for, 202, 223
referencing disk files, 200
Files
binary operations, 208–211
displaying from command line, 223–226
displaying text files, 203–206
file stream objects, 197–199
reading binary data from, 214–217
Files (continued)
- referencing disk files, 199–200
- storing data in, 197
- summary, 217–219
- text files vs. "binary" files, 206–208
- writing binary data to, 211–214
- writing text to, 200–203

fill() function, output streams, 528

fixed, 237–240

fixed stream manipulator, 527

Flags
- file mode flags, 530
- seek direction flags, 530

float data type. See also Floating-point data
- description and range of, 484
- f suffix, 486
- for floating-point data, 21–22
- precision of, 484

FloatFraction class
- complete version of, 442–445
- implementing and testing, 438–440
- inheriting base-class constructors, 441–442
- problems with, 440–441
- protected members of, 445–446
- as subclass of Fraction class, 436–437
- virtual functions in, 454–455

Floating-point data
- comparing integers with, 33–34
- converting strings to, 183
- data types for, 20–22
- decimal point in, 485
- defined, 542–543
- division of, 480
- making a value persistent and recalculated as needed, 453, 459–460
- range and precision of, 484
- reading binary data, 209–210

Flow control, in loops. See also Control structures, 37–38, 48

flush() function, output streams, 528

flush stream manipulator, 527

Folders, referencing disk files in, 199–200

For each. See Range-based for (for each)

for loops
- comparable statements in other languages, 79–80
- comparing with while loops, 71–72
- compound statements and, 74
- counting with, 67–68
- declaring variables on the fly, 74–75
- examples, 70–71
- overview of, 67
- printing numbers from 1 to N with for loop, 72–73
- specifying initializer, condition, and increment values, 68–69
- summary, 80–81
- swap function and, 157–161
- testing for prime numbers, 75–79
- testing randomness with arrays, 126–127
- true/false values in, 80
- using with arrays, 121–123

for statements
- as control structure, 231
- syntax of, 494–495

Fortran, 120

fout object, output streams and, 198–200

Fraction class
- arithmetic functions added to, 300–305
- complete version of, 352–357
- constructors for, 317–320
- copy constructor for, 325–329
- finding GCF of, 291–292
- finding LCM of, 292
- initializing objects from strings, 329–331
- inline functions of, 343–348, 352–357
- overview of, 286–289
- support functions of, 293–296
- testing, 296–300

this keyword used with, 378
Friend functions, 337–338

fstream
   for generic file stream, 198
   including, 201
ftemp variable, for holding Fahrenheit values, 19, 24–25

Functions
   arithmetic functions added to Fraction class, 300–305
      avg(), 87–88
   C-strings, 517–518
   calling, 86–87, 179
   character conversion, 519
   character-testing, 519
   data-conversion, 518
   declaring, 85, 500–501
   defining, 85–86
   Euclid’s algorithm for GCFs, 101–106
   friend functions, 337–338
   GCF function added to Fraction class, 291–292
   inline, 289–291
   LCM function added to Fraction class, 292
   local and global variables and, 93–95
   math, 520
   modules for placing in different files, 234–235
   normalize added to Fraction class, 293–296
   operator. See Operators
   overloading, 227–230
   overview of, 83–84
   prime factorization, 96–100
   prime number, 90–93
   in programs, 89–90
   random number generator, 110–113
   randomized, 521
   recursive, 95–96, 106–110
   single-character, 519
   strftime format, 523–524
   string manipulation, 172–174
   in Subtraction Game, 113–115
   summary, 115–116
   support functions of Fraction class, 296–300
   time functions, 521–523
   virtual. See Virtual functions
   zero_out_array, 165–167

G

GCFs (greatest common factors)
   adding GCF function to Fraction class, 291–292
   defined, 543
   Euclid's algorithm for, 101–103
   how it works, 103–106
Get a Number program, string example, 180–183
get() function, stream input, 528
get_divisors function, 99–100
get_int function, 214
getline() function
   description of, 528
   reading string input, 179–180, 182
   retrieving line of input with, 176
   working with string type and, 192
get_number function, 230

Global functions
   defining operators as, 334
   in Fraction class, 355–356
   operators as, 336–338

Global variables
   declaring functions and, 93–95
   defined, 543
   initializing to zero, 119
   necessity of, 95

go to statements, 497

Graphical-user interfaces (GUIs)
   OOP and, 279
   safe inheritance and, 448–451

Greater than (>) operator, 47, 477
Greater than or equal to (>=) operator
  associativity, precedence, and syntax of, 477
  types of relational operators, 47
  using with for loops, 70
Greatest common factors. See GCFs (greatest common factors)
GUIs (graphical-user interfaces)
  OOP and, 279
  safe inheritance and, 448–451
H
  .h files, C language, 511
Header files
  cstring and string in, 224
  defined, 543
  #include and, 510
hex stream manipulator, 527
Hexadecimal notation (base 16)
  ASCII codes, 514–515
  in C++Ox specification, 485
  decimal and binary equivalents, 147
  storing numeric variables as, 145
High-level languages, comparing C++ with other, 7–8
I
I/O (input/output)
  console stream objects, 525–526
  file I/O functions, 529–530
  functions, 528
  stream classes demonstrating extensibility of OOP, 464–466
  stream manipulators, 526–527
IDEs (integrated development environments)
  defined, 543
  entering program statements with, 8
  #if directive, 508–509
if-else statements
  syntax of, 3, 493
  testing a value against a series of target values, 233
  use with compound statements, 36–38
if statements
  as control structure, 231
  syntax of, 35
  true/false values in, 80
  use with compound statements, 36–38
  #ifdef directive, 510
ifstream
  file-input streams, 198
  including, 201
  turning off file-error exceptions, 240
Implementation, defined, 543
#include directive
  cctype, 185
cstring, 175
  for function declarations, 235
  iostream and fstream, 201
  list templates, 415
  overview of, 510–511
  string type, 189
  supporting specific parts of standard library, 15
  syntax options, 299–300
  using at beginning of Visual Studio programs, 13
Increment (++) operator
  associativity, precedence, and syntax of, 476
  precedence of, 166
  reducing keystrokes with, 67
  using with loops, 44–45, 51–52, 68–70
Incrementing
  in for loops, 68–69
  in while loop, 44–45
Indexes
  boundary problems in, 141
  comparing pointer references with, 163
  data by numbers (arrays), 117–118
  defined, 543
  one-based, 120
  zero-based, 119–120
### Indirection
- defined, 543
- indirection-member operator (\(\rightarrow\)), 391
- indirection operator (\(^*\)), 149, 166

### Inequality (\(!=\)) operator
- associativity, precedence, and syntax of, 477
- types of relational operators, 47

### Infinite loops
- defined, 544
- for exiting files, 205
- overview of, 46

### Infix (Standard) notation, 423

### Inheritance
- of base class constructors (C++0x only), 441–442
- defined, 544
- in FloatFraction class, 442–445
- hierarchies for safe, 448–451
- implementing and testing in
  - FloatFraction class, 438–440
- interfaces implemented via, 462
- object containment and, 447–448
- overview of, 435
- problems in FloatFraction class, 440–441
- protected members and, 445–446
- public keyword for qualifying base classes, 437–438
- subclassing, 435–437
- summary, 451–452
- virtual functions and, 461

### Initialization. See also Constructors
- of arrays, 119
- assigning variable values during, 49
- consistency of, 314–315
- of Fraction objects from strings, 329–331
- initializer values in for loops, 68–69
- of types while declaring, 307

### Inline functions
- addition (+) operator as, 340
- cannot be virtual, 455
- constructors as, 308
- defined, 544
- of Fraction class, 289–291
- function modifiers, 501

### Input stream functions. See also I/O (input/output), 528
- insert function, member functions, 421

### Instance/instantiation
- abstract classes cannot be used for, 462–463
- defined, 544
- comparing pointer values with int variables, 151–152
- converting strings to, 183
- "cooked" literals and, 359
- description and range of, 484
- natural integer type, 245–246
- pointer to, 148
- range compared with double, 127
- reading binary data, 209–210
- swapping numbers with, 485
- swapping values of two int variables, 155–156
- syntax of, 34

### Integer data
- address expressions and, 162
- char type and, 171
- comparing with floating point data, 33–34
- converting strings to, 183
- data types, 20–21
- defined, 544
- division of, 480
- get_int function, 214
- reading binary data, 209–210
- short, long, and long long, 244–245
- signed and unsigned, 245
- suffixes in representation of, 486
- two's complement format for signed integers, 487–489
Interfaces
abstract classes for specifying and enforcing a set of services, 463–464
defined, 544
implementing through class inheritance, 462
relationship between GUIs and OOP, 279
safe inheritance in GUIs, 448–451

ios::app, file mode flags, 530
ios::ate, file mode flags, 530
ios::beg, seek direction flags, 530
ios::binary, file mode flags, 213, 530
ios::cur, seek direction flags, 530
ios::end, seek direction flags, 530
ios::in, file mode flags, 216, 530
ios::out, file mode flags, 213, 530
ios::trunc, file mode flags, 530

is_open(), description of file I/O functions, 529

Iteration. See also Loops
applied to Fibonacci numbers, 250
creating iterators for List template, 416–418, 534
declared, 544
iterative approach to linked list, 400
pointers compared with iterators, 418
recursion compared with, 401–402

Iterators, 416-418, 534

J
join (,) operator
associativity, precedence, and syntax of, 477
as delimiters in text, 187
uses of, 482

K

Keywords
defined, 545
program syntax and, 3
use by high-level languages, 7
variable names and, 29

L
L suffix, representing long int format, 486
Last-in-first-out. See LIFO (last-in-first-out)
Late binding, resolving address of virtual functions at runtime, 472
LCM (lowest common multiple)
adding LCM function to Fraction class, 292
computing from greatest common factors (GCFs), 101
defined, 545
Left shift and assign (<<=) operator, 477
left stream manipulator, 527
Less than (<) operator
associativity, precedence, and syntax of, 477
ordered lists and, 419
types of relational operators, 47
Less than or equal to (<=) operator, 47, 477
LF (linefeed), 513–514
Libraries, C++
math library, 57
standard library, 15
STL (Standard Template Library). See STL (Standard Template Library)
LIFO (last-in-first-out)
defined, 545
function calls, 96
stacks, 425

#line directive, 511

Linked lists
alphabetical, 393–395
implementing, 391–393
list template for, 413
memory leaks in, 399–400
node design for, 390–391
overview of, 389
printing out names in alpha order, 395–399
smart pointers for cleaning up (C++Ox only), 400–401
summary, 411–412
Linkers, building a C++ program, 8–9
List classes (<list>)
- creating, 415–416
- in STL, 533–534
- syntax of, 415
List template
- continual sorting, 421–422
- creating iterators for, 416–418
- creating List class for, 415–416
- overview of, 413–414
- range-based for (for each) used in place of iterators, 418
- writing ordered list program, 419–421
Lists, as container for range-based for, 257
Literals
- 64-bit literals (constants), 246–247
- cooked literals, 359
- data types of, 485–486
- defined, 545
- overview of, 357
- raw string literals, 244, 273, 358
- user-defined, 330–331, 464
LL suffix, representing long long format, 486
Local variables
- declaring functions and, 93–95
- defined, 545
Localizing numbers, 254–255
Logical (Boolean) operators
- determining what is true, 55
- precedence of, 54
- testing a person’s age (example of Boolean logic), 55–56
- type of, 53–54
Logical negation (!) operator
- associativity, precedence, and syntax of, 476
- swap function and, 161
- types of Boolean operators, 54
long double int, 484
long int
- 32-bit, 244–245
description and range of, 484
I suffix for, 486
long long int
- 64-bit, 244–245
- challenges of, 247–248
- C++Ox specification supporting, 34, 244–246
- description and range of, 484
- formatting long long numbers, 248–249
- IL suffix for, 486
- overview of, 243
- syntax of, 245
Loops
- 2-D arrays and, 142–143
- in Card Dealer #3, 139
- counting with, 67–68, 545
- defined, 545
- do while, 230–232
- for. See for loops
- increment operator (++) used in, 51–52
- infinite, 46, 205
- in prime factorization, 99–100
- print 1 to N example, 46–49
- testing randomness with, 126–127
- using strtok function with, 188
- using swap function for sorting, 157–161
- using with arrays, 117, 121–123
- while loops, 43–45
Lowest common multiple. See LCM (lowest common multiple)
Lukasiewicz, Jan, 424
Lvalue, 545
M
Machine code
- array indexing and, 120
- defined, 5, 545–546
- high-level languages and, 7
- native language of computer (1s and 0s), 3
- translating statements into, 170–171
Magic numbers, minimizing appearance of, 254
main function
   defined, 546
   prompting for filenames, 202
Main memory (RAM)
   defined, 546
   nonpersistence of, 197
Math library (cmath)
   accessing, 57
   functions in, 520
   including, 181
Matrix, 2-D arrays and, 142–143
MAX_PATH, setting for filename length, 202
Member functions
   constructors as, 307
   declaring for Point class, 282–283
   defined, 546
   destructors as, 370
   of Fraction class, 287
   friend functions and, 337–338
   inlining, 290–291
   in List classes (<list>), 534
   operators as, 333–334
   overriding, 453–454
   push_back, 415
   push_front, 416
   returning existing objects vs. returning new objects, 382
   sort, 421
   in Stack classes (<stack>), 536
   in string class (<string>), 532–533
   syntax of, 283
Members
   assigning values to class data fields, 280
   defined, 541, 546
   initializing within a class (C++Ox only), 309–310
   private, 281–284
   protected, 445–446
   public, 279
   relationship with objects and classes, 284
   restricting access to, 287
   structures and, 281
Memory
   allocating blocks of, 366–367
   defined, 546
   dynamic allocation. See Dynamic memory allocation
   RAM (random access memory), 197
Memory leaks
   in linked lists, 399–400
   overview of, 368
Microsoft Visual Studio, 12–13
Minus (-) operator. See Change sign of (-) operator
Modules
   advantages of multiple, 235–236
   defined, 546
   overview of, 234
   source files as, 241
Modulus or remainder operator (%)
   associativity, precedence, and syntax of, 476
   declaring, 335–336
   prime number function and, 92
   random number generation and, 113
   using in Odd-or-Even program, 41–42
mult function
   adding arithmetic functions to Fraction class, 300–305
   refining in Fraction class, 347–348
Multi-dimensional arrays, 142–143
Multiple constructors (overloading), 309
Multiple modules, 234–236
Multiply and assign (=) operator, 477
Multiply (*) operator, 476
MyStack class, Tower of Hanoi example
   creating, 403–404
   using, 404–405
N
   NAK (no acknowledgment), 513–514
Namespaces, std namespace, 414
Nesting
   defined, 546
   nested loops, 160
new operator
- allocating memory blocks with, 366–367
- associativity, precedence, and syntax of, 476
- creating node with, 391
- dealing with problems in memory allocation, 368
- dynamic memory allocation and, 363–364
- example allocating array of any size, 369–370
- pointers to objects and, 365–366
- syntax of, 315

Newline character
- for advancing to next print line, 16–18
- defined, 546

NIM (Subtraction Game)
- decision making example, 60–63
- function for enhancing, 113–115

noboolalpha stream manipulator, 527

Nodes, designing for linked lists, 390–391

Normalize function
- adding support functions to Fraction class, 293–296
- overriding, 453–454, 457, 459–460
- set function calling, 350

noshowbase stream manipulator, 527
noshowpoint stream manipulator, 527

NOT (!) operator. See Logical negation (!) operator

nounitbuf stream manipulator, 527
nouppercase stream manipulator, 527

Null-terminated strings, 189

NULL values
- NUL character code, 513–514
- random number generator and, 112
- use with strtok() function, 187
- use with time() function, 112

nullptr keyword
- in C++Ox specification, 112, 187

linked lists and, 390–391
- setting null pointers with, 244
- uses of, 262–263

num
- protected members of FloatFraction class, 445–446
- variable names and, 28

Numbers
- base 10. See Decimal notation (base 10)
- base 16. See Hexadecimal notation (base 16)
- base 2. See Binary notation (base 2)
- converting numeric values into text characters, 169–170
- counting, 67–68
- Fibonacci numbers, 250–254
- function for prime factorization, 96–100
- function for prime numbers, 90–93
- Get a Number program (example), 180–183
- get_number function, 181
- localizing, 254–255
- lowest divisor as prime number, 100–101
- numeric data types, 20–21
- numeric expressions in single quotes, 186
- printing from 1 to N with for loop, 72–73
- printing from 1 to N with while loop, 46–49
- random number generator, 110–113
- storing, 485
- testing for prime numbers with for loops, 75–79
- testing for prime numbers with while loops, 57–60

O

Object containment, as alternative to inheritance, 447–448
Object independence. See Polymorphism

Objects
- built-in behaviors, 453
- class destructors, 370–371
- in classes, 277–278
- console stream objects, 525–526
- defined, 178–179, 546
- as individual data items within classes, 277
- initializing from strings, 329–331
- pointers to, 365–366
- quasi-intelligence of, 278
- relationship with classes and members, 284
- `this` keyword as pointer to current object, 378

Oct stream manipulator, 527
Octal notation (base 8), 485
Odd-or-Even program, 39–42
Offsets, arrays as measure of, 119–120
ofstream (file-output), 198–200
One-based indexing, 120, 547
OOP (object-oriented programming)
- advantages of, 278
- classes in, 277
- comparing C++ with C, 7
- defined, 547
- history of, 278–279
- linked list example. See Linked lists
- multiple modules and, 236
- object independence. See Polymorphism
- overloading and, 228
- stream classes demonstrating extensibility of, 464–466
- system orientation of, 472
- Tower of Hanoi animation example. See Tower of Hanoi, animated
- using objects without knowing type or what function it calls, 470–471

operator+ function, writing for String class, 381–386
Operators. See also by individual type
- associativity, precedence, and syntax of, 476–477
- declaring, 334–336
- defined, 547
- for Fraction class, 343–348, 352–357
- as global functions, 336–338
- overloading, 228, 348–349
- for Point class, 340–343
- print function for class, 351–352
- in Random-Access Write example, 213
- references used with, 338–340
- summary, 360–362
- syntax of, 333
- user-defined literals (C++Ox only), 357–359

Optimization, of programs, 26–27
OR (||) operator
- in Subtraction Game example (NIM), 63
- types of Boolean operators, 54

Ordered list program, 419–421
ostream class, 351–352
Output stream functions. See also I/O (input/output), 528
Overloading
- constructors, 309
- defined, 547
- functions, 227–228
- object orientation and, 228
- operators, 335, 348–349
- print_array function (example), 228–230

override keyword, 461
Overriding
- member functions, 453–454
- normalize function, 453–454, 457, 459–460
- requiring explicit (C++Ox only), 460–461
smart pointers (C++Ox only), 400–401
string manipulation functions and, 172
summary of rules regarding, 168
swap function and, 155–156
swap function for sorting arrays, 156–161
deleted keyword, 378
zero_out_array function, 165–167
Polish Notation. See also RPN (Reverse Polish Notation), 424
Polymorphism
abstract classes, 462–464
benefits of OOP, 278
defined, 548
incomplete type information and, 228
making a floating point value persistent and recalculating as needed, 459–460
overriding member functions and, 453–454
Printable class example, 466–470
pure virtual functions, 461–462
requiring explicit overrides (C++Ox only), 460–461
revised FloatFraction class using virtual functions, 456–458
stream classes demonstrating extensibility of OOP, 464–466
summary, 472–473
system orientation of OOP and, 472
trade off in use of virtual function calls, 455–456
using object without knowing type or what function it calls, 470–471
virtual functions in FloatFraction class, 454–455
Precedence
defined, 548
in evaluation of expressions, 166–167
of operators, 54, 476
Precision of data types, 484
precision(), output stream function, 528
Predefined constants
  advantages of, 226
  list of, 512
  use of, 225
Preprocessor directives
  ## (concatenation) operator, 507
  #define. See #define directive
defined function, 507
definition of, 542
  #elif, 507–508
  #endif, 508
  #error, 508
  #if, 508–509
  #ifdef, 510
  #include. See #include
  #line, 511
overview of, 505
  #undef, 511
Prime factorization function, 96–100
Prime numbers
  function for, 90–93
  lowest divisor as, 100–101
  testing for with for loops, 75–79
testing for with while loops, 57–60
Print function, as class operator, 351–352
Printable class example, true polymorphism in, 466–470
Print_array function, example of function overloading, 228–230
Printing
  advancing to next print line (newline), 16–18
  array elements, 121–123
building ability to print into objects, 465
multiple line programs, 16–18
names in alpha order, 395–399
numbers from 1 to N with for loop, 72–73
numbers from 1 to N with while loop, 46–49
printing a message (example), 11–14
private keyword
  access levels in C++, 446
  in class declaration, 502
Private members
  access levels in C++, 446
  of Fraction class, 287
  friend functions and, 337–338
  of Point class, 281–284
Private/public distinction, encapsulation and, 236
Procedures. See Functions
Processors. See CPUs (central processing units)
Program-logic errors
  exception handling, 237
  testing for, 9–10
Programming
  computer do only what you tell them, 1
determining what a program will do, 1–2
  exercises, 15
  printing a message (example), 11–14
  writing C++ statements, 2–3
Programming, advanced
  advantages of predefined constants, 226
  command-line arguments, 221–222
displaying files from command line, 223–226
do_while loops, 230–232
  exception handling, 237–240
  multiple modules, 234–236
  overloading functions, 227–228
  overview of, 221
  print_array function (example), 228–230
  summary, 240–242
  switch case statements, 232–234
Programs
  building a C++ program, 8–10
  comments in, 23–24
  compared with applications and code, 5
decision making in, 34–35
defined, 6, 548
determining what a program will do, 1–2
as list of things for computer to do, 1
optimization of, 26–27
printing multiple line programs, 16–18
Projects, Microsoft Visual Studio, 12–13

**protected** keyword

in class declaration, 502
overriding functions and, 455

Protected members, of FloatFraction class, 445–446

Prototypes

avg() example, 87–88
calling functions and, 235
declaring functions, 85
declaring global functions, 336
defined, 548
ending with semicolon, 119
of Fraction class, 288–289
syntax of, 501

Pseudocode, 3

Pseudorandom sequences, 112

**public** keyword

access levels in C++, 446
in class declaration, 502
specifying access level for base classes, 437–438

Public members

access levels in C++, 446
declaring classes and, 280
of Fraction class, 287
of structures, 281

Public/private distinction, encapsulation and, 236

Pure virtual functions. See also Virtual functions, 461–462, 548

**push** function, 433

**push_back** member function, 415

**push_front** member function, 416

**put( )**, output stream function, 528

**putback( )**, input stream function, 528

**Q**

Quotation marks (" ")
distinguishing strings from individual characters, 185
in **include** syntax, 510

**R**

RAM (random access memory)
defined, 546
nonpersistence of, 197

**rand** function

in Card Dealer #2, 134
for random number generation, 112–113
testing randomness with arrays, 123–127

Random access

Random-Access Read example, 214–217
Random-Access Write example, 211–214

Random access memory (RAM)
defined, 546
nonpersistence of, 197

Random number generator, 110–113

Randomized functions, in standard library, 521

Randomness, testing with arrays, 123–127

Range-based for (for each)
defined, 243
overview of, 256–258
setting array using, 258–261
using with List template, 418

Ranges of data types, 484

Rational number class. See Fraction class

Raw string literals

in C++Ox specification, 273
defining, 358
overview of, 244

**read( )**
description of input stream functions, 528
input and output of binary data and, 210
readfile commands, 224
Reading
- reading binary data from files, 214–217
- reading string input, 178–180

Records, finding by number, 213–214

Recursion
- applying to Fibonacci numbers, 250
- defined, 548
- function calling itself, 95–96
- iteration compared with, 401–402
- prime factorization, 96–100
- Tower of Hanoi puzzle, 106–110, 408

References
- copy constructors and, 325
- defined, 548–549
- operators used with, 338–340
- reference arguments, 322–323
- reference variables, 321–322

register variable modifier, 499–500
reinterpret_cast operator, 209

Relational operators
- Boolean operators, 50
- precedence of, 54
- returning true or false values, 47

return statements
- returning values with, 18
- syntax of control structures, 497
- transferring control out of loop or function, 231

Return *this statement, 379–380

Return values
- compared with pointers, 321
- declaring functions, 85
- defining functions, 85–86
- from functions, 83

Reusable code
- inheritance and, 435
- object containment and, 447–448

Reverse Polish Notation. See RPN (Reverse Polish Notation)

right shift and assign (>>=) operator, 477
right stream manipulator, 527

Rock, Paper, Scissors game, 115, 267–272
RPN (Reverse Polish Notation)
- designing calculator for, 422–424
- history of Polish Notation, 424
- RPN program, 428–432
- using stack for, 424–426
- using STL stack for, 427–428

Runtime errors, 237

S

Scaling, pointers and, 162
scientific, description of stream manipulators, 527
Scientific notation, 485
Scope
- associativity, precedence, and syntax of, 476
- defined, 549
- uses of Scope (::) operator, 478
Seek direction flags, file I/O functions, 530
seekg(), file I/O function, 529
seekp(), file I/O function, 529
seekp function, for moving file pointer, 214, 216

Set function, normalize function called by, 350
Shallow copying, 376–377
shared_ptr keyword, for smart pointers, 400–401

short
- 16-bit integers (short), 244–245
- description and range of, 484
Short-circuit logic, logical (Boolean) operators and, 54–55
showbase stream manipulator, 527
showpoint stream manipulator, 527
showpos stream manipulator, 527
signed char, 484
signed int
- overview of, 247
- two's complement format for, 487–489
- types of integer data, 245
Signed numbers, representation of, 487-489
Single-character functions, in standard library, 519
Single-precision types. See float
Single quotes (""'), distinguishing strings from individual characters, 185
sizeof operator
  associativity, precedence, and syntax of, 476
  returning size of specified type, variable, or array, 211
  uses of, 478–479
Smart pointers (C++0x only), 400–401
sort
  as power member function, 421
  sorting lists, 421–422
Source code
  compiling as machine code, 5
  defined, 6
  storing as text file, 170
Source files
  defined, 549
  modules and, 235, 241
Space penalty, in use of virtual functions, 455–456
Spaces, as delimiters in text, 187
Spaghetti code, 497
sqrt function
  in math library, 57
  testing for prime numbers with for loops, 75–79
  testing for prime numbers with while loops, 57–60
Square roots, Get a Number program (example), 180–183
Stack classes (<stack>), in STL, 425, 427–428, 535–536
Stacks
  defined, 549
  of function calls, 96
Standard (infix) notation, 423
Standard library
  character conversion functions, 519
  character-testing functions, 519
  data-conversion functions, 518
  math functions, 520
  overview of, 15
  randomized functions, 521
  single-character functions, 519
  str functions, 175
strftime function, formats of, 523–524
strtok (string token) function, 186–188
time function, 521–523
Statements. See also by individual type
  compound. See Compound statements (blocks)
  defined, 6, 549
  entering program statements, 8
  expressions compared with, 52–53
  functions grouping related, 83
  replacing control structures with statement blocks, 231
  switch case statements, 232–234
  syntax of, 492
  translating into machine code, 170–171
  writing C++ statements, 2–3
static
  function modifiers, 501
  variable modifiers, 499–500
Static storage class, 549
Static_cast conversion, 265–266
std namespace, 414
STL (Standard Template Library)
  continual sorting of list, 421–422
  creating iterators for list, 416–418
  creating List class, 415–416
  defined, 549
  List classes (<list>), 533–534
  list template, 413–414
  object-orientation of, 279
STL (Standard Template Library) (continued)
- overview of, 413
- range-based for (for each) used in place of iterators, 418
- Reverse Polish Notation (RPN) calculator, 422–424
- RPN program, 428–432
- stack classes (<stack>), 425, 427–428, 535–536
- string (C-strings) functions, 517–518
- string class (<string>), 531–533

Storage
- controlling using enum classes, 266–267
- of data in files, 197
- Storage classes, 550

String class (C-strings)
- complete version of, 382–386
- deep copying, 377
- shallow copying, 376–377

String concatenation
- defined, 551

String class (<string>), 189–191, 531–533

Stream classes
- demonstrating extensibility of OOP, 464–466
- Printable class example, 466–470

Streams
- console input. See cin
- console output. See cout
- console stream objects, 525–526
- file I/O functions, 529–530
- input stream functions, 528
- manipulators, 526–527
- output stream functions, 528

Strftime function, in standard library, 523–524

Strings (C-strings)
- arrays in Card Dealer #1, 129–132
- arrays of, 128–129
- based on char type, 171–172
- building, 174–177
- building with STL string type, 191–193
- comparing (strcmp), 375
- Convert to Uppercase program (example), 183–185
- defined, 539, 550
- distinguished from individual characters, 185–186
- escape sequences, 177–178
- format and escape sequences, 486–487
- functions for manipulating, 172–174
- functions in standard library, 517
- Get a Number program (example), 180–183
- including, 175, 181
initializing Fraction objects from, 329–331
overview of, 169
reading string input, 178–180
storing text on computers, 169–170
string literals and string variables, 128
strtok function for breaking up input, 186–188
summary, 194–195
of text, 18–19
text string data, 20–21
Strings (<string>)
building strings with, 191–193
declaring variables of, 189–190
input and output and, 191
new in C++, 189
new STL string class, 189–191
other operations on, 193–194
other operations on STL string type, 193–194
working with variables of, 190–191
strlen function, returning string length, 172
strncat function, concatenation of strings, 172, 174
strncpy function, copying strings, 172, 174
Strongly typed enumerations, 244, 263–265
Strousup, Bjarne, 243, 279
strtok (string token) function
for breaking up text input, 186–188
returning null pointer, 330
setting using nullptr keyword, 263
struct keyword
declaring structures, 281
issues with, 312–313
Structures, classes compared with, 281
Subclasses. See also Inheritance
abstract classes and, 463–464
declaring, 279
defined, 550
how to subclass, 435–437
not inheriting constructors from base class, 440
principle of passing something specific
(subclass) to something more general
(base class), 466
virtual functions and, 455
Subroutines. See Functions
substring (substr) class, 248–249
Subtraction (-) operator
adding to Point class, 342–343
associativity, precedence, and syntax of, 476
Subtraction Game (NIM)
decision making example, 60–63
function for enhancing, 113–115
Suffixes
operators, 358–359
in representation of integer values, 486
Swap function
overloading, 227–228
sorting arrays with, 156–161
swapping values of two int variables, 155–156
switch-case statements
in advanced programming, 232–234
syntax of, 495–496
Symbols
automating assignment of symbolic constants, 264–265
defined, 550
enum classes and, 266, 503
literals compared with, 485
Syntax
ergors, 9–10, 237
summary of, 491-504
Systems
OOP and, 472
polymorphism and, 465
T
tellg() function, file I/O, 529
tellp() function, file I/O, 529
temp variable, 155–156
Temperature, converting Celsius to/from Fahrenheit, 22–26

Templates. See also STL (Standard Template Library)
defined, 550
as generalized classes, 413
writing own, 414–415

Testing
Fraction class, 296–300
Point class, 284–286
for prime numbers, 57–60
programs, 9–10
randomness with arrays, 123–127

Text
delimiters in, 186
displaying text files, 203–206
storing on computers, 169–170
strings of, 18–19
strtok function for breaking up input, 186–188
text files vs. binary files, 206–208
text files vs. "binary" files, 206–208
writing to files, 200–203

Text editors
displaying text files, 203–206
entering program statements with, 8
viewing file contents, 201

Text string data. See Strings (C-strings)
this keyword
as pointer to current object, 378
return *this statement as last statement in assignment operator function, 379–380

throw statements
exception handling and, 239–240
syntax of control structures, 497

Time functions, in standard library, 521–523

Tokens, substrings, 186
tolower (c) function, 184–185
toupper (c) function, 184–185

Tower of Hanoi, animated version
building, 405–411
creating MyStack class, 403–404
overview of, 402–403
summary, 411–412
using MyStack class, 404–405

Tower of Hanoi, basic version, 106–110

Trigonometric functions, 520

True/false. See Boolean values (true/false)
try keyword, exception handling and, 238–240

Two's complement
defined, 551
format for signed integers, 487–489
integers and, 33

Type casts. See Casts
Type checking, in OOP, 471
Types. See Data types

U
U suffix, representing unsigned int format, 486
ULL suffix, representing unsigned long long format, 486
#undef directive, 511
Unicode, 170
unitbuf, stream manipulator, 527
unsigned char, 484
unsigned int
description and range of, 484
overview of, 247
types of integer data, 245
u suffix, 486

unsigned long, 484
unsigned long long, 484, 486
unsigned short, 484

uppercase, stream manipulator, 527

User-defined functions, 85
User-defined literals (C++Ox only), 357–359, 464
Users, defined, 6, 542
using statement
  accessing objects with, 16
  working with string type and, 192

V

Variables
  assigning values to pointer variables, 161–163
  content vs. address of, 151–152
  declaring, 498–500
  declaring on the fly, 74–75
  declaring prior to use, 20
  declaring with auto keyword, 261–262
  defined, 551
  doubling with double_it function, 152–155
  local and global variables, 93–95
  naming rules and conventions, 28–29
  sharing, 235
  storing data with, 19–20
  string variables, 128, 189–191
  swapping values of two int variables, 155–156
Virtual functions
  in abstract classes, 462–463
  declaring member functions that might be overridden virtual, 454–455
  defined, 551
  late binding, 472
  pure virtual functions, 461–462
  restrictions on use of, 455
  revised FloatFraction class using, 456–458
  trade offs in use of, 455–456
virtual keyword, as function modifiers, 454–455, 501
Visual Basic, 7

volatile, variable modifier, 499–500
vtable pointer, for calling virtual functions, 456

W

wcerr, stream object, 526
wchar_t, 484
wcin, stream object, 526
wclog, stream object, 526
wcout, stream object, 526

while loops
  in Card Dealer #3, 140
  iterative approach to linked list, 400
  for loops compared with, 71–72
  overview of, 43
  print 1 to N loop example, 46–49
  pseudo code for printing numbers from 1 to N, 43–45
  syntax of, 69
  true/false values in, 80
  using strtok function with, 188
while statements, as control structure, 231, 493–494
width() function, stream output, 528
Word processors, 201
write() function
description of output stream functions, 528
input and output of binary data and, 210
Writing
  binary data to files, 211–214
  text to files, 200–203

Z

Zero-based indexes
  arrays and, 119–120
  defined, 551
Zero_out_array function, 165–16