C++11 for Programmers
Second Edition
Deitel® Developer Series

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.
Trademarks

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

Microsoft, Visual Studio and the Windows logo are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Throughout this book, trademarks are used. Rather than put a trademark symbol in every occurrence of a trademarked name, we state that we are using the names in an editorial fashion only and to the benefit of the trademark owner, with no intention of infringement of the trademark.
To our review team:

Dean Michael Berris
Danny Kalev
Linda M. Krause
James P. McNellis
Robert C. Seacord
José Antonio González Seco

We are grateful for your guidance and expertise.

Paul and Harvey Deitel
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.12</td>
<td>Functions with Empty Parameter Lists</td>
<td>162</td>
</tr>
<tr>
<td>6.13</td>
<td>Inline Functions</td>
<td>163</td>
</tr>
<tr>
<td>6.14</td>
<td>References and Reference Parameters</td>
<td>164</td>
</tr>
<tr>
<td>6.15</td>
<td>Default Arguments</td>
<td>167</td>
</tr>
<tr>
<td>6.16</td>
<td>Unary Scope Resolution Operator</td>
<td>169</td>
</tr>
<tr>
<td>6.17</td>
<td>Function Overloading</td>
<td>170</td>
</tr>
<tr>
<td>6.18</td>
<td>Function Templates</td>
<td>173</td>
</tr>
<tr>
<td>6.19</td>
<td>Recursion</td>
<td>175</td>
</tr>
<tr>
<td>6.20</td>
<td>Example Using Recursion: Fibonacci Series</td>
<td>179</td>
</tr>
<tr>
<td>6.21</td>
<td>Recursion vs. Iteration</td>
<td>182</td>
</tr>
<tr>
<td>6.22</td>
<td>Wrap-Up</td>
<td>184</td>
</tr>
<tr>
<td>7</td>
<td>Class Templates array and vector; Catching Exceptions</td>
<td>185</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>186</td>
</tr>
<tr>
<td>7.2</td>
<td>arrays</td>
<td>186</td>
</tr>
<tr>
<td>7.3</td>
<td>Declaring arrays</td>
<td>188</td>
</tr>
<tr>
<td>7.4</td>
<td>Examples Using arrays</td>
<td>188</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Declaring an array and Using a Loop to Initialize the array’s Elements</td>
<td>188</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Initializing an array in a Declaration with an Initializer List</td>
<td>189</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Specifying an array’s Size with a Constant Variable and Setting array Elements with Calculations</td>
<td>190</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Summing the Elements of an array</td>
<td>192</td>
</tr>
<tr>
<td>7.4.5</td>
<td>Using Bar Charts to Display array Data Graphically</td>
<td>193</td>
</tr>
<tr>
<td>7.4.6</td>
<td>Using the Elements of an array as Counters</td>
<td>195</td>
</tr>
<tr>
<td>7.4.7</td>
<td>Using arrays to Summarize Survey Results</td>
<td>196</td>
</tr>
<tr>
<td>7.4.8</td>
<td>Static Local arrays and Automatic Local arrays</td>
<td>198</td>
</tr>
<tr>
<td>7.5</td>
<td>Range-Based for Statement</td>
<td>200</td>
</tr>
<tr>
<td>7.6</td>
<td>Case Study: Class GradeBook Using an array to Store Grades</td>
<td>202</td>
</tr>
<tr>
<td>7.7</td>
<td>Sorting and Searching arrays</td>
<td>209</td>
</tr>
<tr>
<td>7.8</td>
<td>Multidimensional arrays</td>
<td>211</td>
</tr>
<tr>
<td>7.9</td>
<td>Case Study: Class GradeBook Using a Two-Dimensional array</td>
<td>214</td>
</tr>
<tr>
<td>7.10</td>
<td>Introduction to C++ Standard Library Class Template vector</td>
<td>221</td>
</tr>
<tr>
<td>7.11</td>
<td>Wrap-Up</td>
<td>227</td>
</tr>
<tr>
<td>8</td>
<td>Pointers</td>
<td>228</td>
</tr>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>229</td>
</tr>
<tr>
<td>8.2</td>
<td>Pointer Variable Declarations and Initialization</td>
<td>229</td>
</tr>
<tr>
<td>8.3</td>
<td>Pointer Operators</td>
<td>231</td>
</tr>
<tr>
<td>8.4</td>
<td>Pass-by-Reference with Pointers</td>
<td>233</td>
</tr>
<tr>
<td>8.5</td>
<td>Built-In Arrays</td>
<td>238</td>
</tr>
<tr>
<td>8.6</td>
<td>Using const with Pointers</td>
<td>240</td>
</tr>
<tr>
<td>8.6.1</td>
<td>Nonconstant Pointer to Nonconstant Data</td>
<td>241</td>
</tr>
<tr>
<td>8.6.2</td>
<td>Nonconstant Pointer to Constant Data</td>
<td>241</td>
</tr>
</tbody>
</table>
9 Classes: A Deeper Look; Throwing Exceptions 256
9.1 Introduction 257
9.2 Time Class Case Study 258
9.3 Class Scope and Accessing Class Members 264
9.4 Access Functions and Utility Functions 265
9.5 Time Class Case Study: Constructors with Default Arguments 266
9.6 Destructors 272
9.7 When Constructors and Destructors Are Called 272
9.8 Time Class Case Study: A Subtle Trap—Returning a Reference or a Pointer to a private Data Member 276
9.9 Default Memberwise Assignment 279
9.10 const Objects and const Member Functions 281
9.11 Composition: Objects as Members of Classes 283
9.12 friend Functions and friend Classes 289
9.13 Using the this Pointer 291
9.14 static Class Members 297
9.15 Wrap-Up 302

10 Operator Overloading: Class string 303
10.1 Introduction 304
10.2 Using the Overloaded Operators of Standard Library Class string 305
10.3 Fundamentals of Operator Overloading 308
10.4 Overloading Binary Operators 309
10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators 310
10.6 Overloading Unary Operators 314
10.7 Overloading the Unary Prefix and Postfix ++ and -- Operators 315
10.8 Case Study: A Date Class 316
10.9 Dynamic Memory Management 321
10.10 Case Study: Array Class 323
10.10.1 Using the Array Class 324
10.10.2 Array Class Definition 328
10.11 Operators as Member vs. Non-Member Functions 336
10.12 Converting Between Types 337
10.13 explicit Constructors and Conversion Operators 338
10.14 Overloading the Function Call Operator () 340
10.15 Wrap-Up 341
## 11 Object-Oriented Programming: Inheritance

11.1 Introduction

11.2 Base Classes and Derived Classes

11.3 Relationship between Base and Derived Classes

   11.3.1 Creating and Using a CommissionEmployee Class
   11.3.2 Creating a BasePlusCommissionEmployee Class Without Using Inheritance
   11.3.3 Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy
   11.3.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Data
   11.3.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Data

11.4 Constructors and Destructors in Derived Classes

11.5 public, protected and private Inheritance

11.6 Software Engineering with Inheritance

11.7 Wrap-Up

## 12 Object-Oriented Programming: Polymorphism

12.1 Introduction

12.2 Introduction to Polymorphism: Polymorphic Video Game

12.3 Relationships Among Objects in an Inheritance Hierarchy

   12.3.1 Invoking Base-Class Functions from Derived-Class Objects
   12.3.2 Aiming Derived-Class Pointers at Base-Class Objects
   12.3.3 Derived-Class Member-Function Calls via Base-Class Pointers
   12.3.4 Virtual Functions and Virtual Destructors

12.4 Type Fields and switch Statements

12.5 Abstract Classes and Pure virtual Functions

12.6 Case Study: Payroll System Using Polymorphism

   12.6.1 Creating Abstract Base Class Employee
   12.6.2 Creating Concrete Derived Class SalariedEmployee
   12.6.3 Creating Concrete Derived Class CommissionEmployee
   12.6.4 Creating Indirect Concrete Derived Class BasePlusCommissionEmployee
   12.6.5 Demonstrating Polymorphic Processing

12.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”

12.8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, dynamic_cast, typeid and type_info

12.9 Wrap-Up

## 13 Stream Input/Output: A Deeper Look

13.1 Introduction
Contents

13.2 Streams 417
  13.2.1 Classic Streams vs. Standard Streams 417
  13.2.2 istream Library Headers 418
  13.2.3 Stream Input/Output Classes and Objects 418
13.3 Stream Output 420
  13.3.1 Output of char * Variables 421
  13.3.2 Character Output Using Member Function put 421
13.4 Stream Input 422
  13.4.1 get and getline Member Functions 422
  13.4.2 istream Member Functions peek, putback and ignore 425
  13.4.3 Type-Safe I/O 425
13.5 Unformatted I/O Using read, write and gcount 425
13.6 Introduction to Stream Manipulators 426
  13.6.1 Integral Stream Base: dec, oct, hex and setbase 427
  13.6.2 Floating-Point Precision (precision, setprecision) 427
  13.6.3 Field Width (width, setw) 429
  13.6.4 User-Defined Output Stream Manipulators 430
13.7 Stream Format States and Stream Manipulators 431
  13.7.1 Trailing Zeros and Decimal Points (showpoint) 432
  13.7.2 Justification (left, right and internal) 433
  13.7.3 Padding (fill, setfill) 435
  13.7.4 Integral Stream Base (dec, oct, hex, showbase) 436
  13.7.5 Floating-Point Numbers; Scientific and Fixed Notation (scientific, fixed) 437
  13.7.6 Uppercase/Lowercase Control (uppercase) 438
  13.7.7 Specifying Boolean Format (boolalpha) 438
  13.7.8 Setting and Resetting the Format State via Member Function flags 439
13.8 Stream Error States 440
13.9 Tying an Output Stream to an Input Stream 443
13.10 Wrap-Up 443

14 File Processing 444

14.1 Introduction 445
14.2 Files and Streams 445
14.3 Creating a Sequential File 446
14.4 Reading Data from a Sequential File 450
14.5 Updating Sequential Files 456
14.6 Random-Access Files 456
14.7 Creating a Random-Access File 457
14.8 Writing Data Randomly to a Random-Access File 462
14.9 Reading from a Random-Access File Sequentially 464
14.10 Case Study: A Transaction-Processing Program 466
14.11 Object Serialization 473
14.12 Wrap-Up 473
17.2 Example: Handling an Attempt to Divide by Zero 561
17.3 Rethrowing an Exception 567
17.4 Stack Unwinding 568
17.5 When to Use Exception Handling 570
17.6 Constructors, Destructors and Exception Handling 571
17.7 Exceptions and Inheritance 572
17.8 Processing new Failures 572
17.9 Class unique_ptr and Dynamic Memory Allocation 575
17.10 Standard Library Exception Hierarchy 578
17.11 Wrap-Up 579

18 Introduction to Custom Templates 581
18.1 Introduction 582
18.2 Class Templates 582
18.3 Function Template to Manipulate a Class-Template Specialization Object 587
18.4 Nontype Parameters 589
18.5 Default Arguments for Template Type Parameters 589
18.6 Overloading Function Templates 589
18.7 Wrap-Up 590

19 Class string and String Stream Processing: A Deeper Look 591
19.1 Introduction 592
19.2 string Assignment and Concatenation 593
19.3 Comparing strings 595
19.4 Substrings 598
19.5 Swapping strings 598
19.6 string Characteristics 599
19.7 Finding Substrings and Characters in a string 601
19.8 Replacing Characters in a string 603
19.9 Inserting Characters into a string 605
19.10 Conversion to Pointer-Based char * Strings 606
19.11 Iterators 607
19.12 String Stream Processing 609
19.13 C++11 Numeric Conversion Functions 612
19.14 Wrap-Up 613

20 Bits, Characters, C Strings and structs 615
20.1 Introduction 616
20.2 Structure Definitions 616
20.3 typedef 618
20.4 Example: Card Shuffling and Dealing Simulation 618
20.5 Bitwise Operators 621
20.6 Bit Fields 630
20.7 Character-Handling Library 633
20.8 C String-Manipulation Functions 639
20.9 C String-Conversion Functions 646
20.10 Search Functions of the C String-Handling Library 651
20.11 Memory Functions of the C String-Handling Library 655
20.12 Wrap-Up 659

21 Other Topics 660
21.1 Introduction 661
21.2 const_cast Operator 661
21.3 mutable Class Members 663
21.4 namespaces 665
21.5 Operator Keywords 668
21.6 Pointers to Class Members (.* and ->* ) 670
21.7 Multiple Inheritance 672
21.8 Multiple Inheritance and virtual Base Classes 677
21.9 Wrap-Up 681

22 ATM Case Study, Part 1: Object-Oriented Design with the UML 682
22.1 Introduction 683
22.2 Introduction to Object-Oriented Analysis and Design 683
22.3 Examining the ATM Requirements Document 684
22.4 Identifying the Classes in the ATM Requirements Document 691
22.5 Identifying Class Attributes 698
22.6 Identifying Objects’ States and Activities 703
22.7 Identifying Class Operations 707
22.8 Indicating Collaboration Among Objects 714
22.9 Wrap-Up 721

23 ATM Case Study, Part 2: Implementing an Object-Oriented Design 725
23.1 Introduction 726
23.2 Starting to Program the Classes of the ATM System 726
23.3 Incorporating Inheritance into the ATM System 732
23.4 ATM Case Study Implementation 739
  23.4.1 Class ATM 740
  23.4.2 Class Screen 747
  23.4.3 Class Keypad 749
  23.4.4 Class CashDispenser 750
  23.4.5 Class DepositSlot 752
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.4.6</td>
<td>Class Account</td>
<td>753</td>
</tr>
<tr>
<td>23.4.7</td>
<td>Class BankDatabase</td>
<td>755</td>
</tr>
<tr>
<td>23.4.8</td>
<td>Class Transaction</td>
<td>759</td>
</tr>
<tr>
<td>23.4.9</td>
<td>Class BalanceInquiry</td>
<td>761</td>
</tr>
<tr>
<td>23.4.10</td>
<td>Class Withdrawal</td>
<td>763</td>
</tr>
<tr>
<td>23.4.11</td>
<td>Class Deposit</td>
<td>768</td>
</tr>
<tr>
<td>23.4.12</td>
<td>Test Program ATMCaseStudy.cpp</td>
<td>771</td>
</tr>
<tr>
<td>23.5</td>
<td>Wrap-Up</td>
<td>771</td>
</tr>
</tbody>
</table>

### A Operator Precedence and Associativity | 774

### B ASCII Character Set | 777

### C Fundamental Types | 778

### D Number Systems | 780

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1</td>
<td>Introduction</td>
<td>781</td>
</tr>
<tr>
<td>D.2</td>
<td>Abbreviating Binary Numbers as Octal and Hexadecimal Numbers</td>
<td>784</td>
</tr>
<tr>
<td>D.3</td>
<td>Converting Octal and Hexadecimal Numbers to Binary Numbers</td>
<td>785</td>
</tr>
<tr>
<td>D.4</td>
<td>Converting from Binary, Octal or Hexadecimal to Decimal</td>
<td>785</td>
</tr>
<tr>
<td>D.5</td>
<td>Converting from Decimal to Binary, Octal or Hexadecimal</td>
<td>786</td>
</tr>
<tr>
<td>D.6</td>
<td>Negative Binary Numbers: Two’s Complement Notation</td>
<td>788</td>
</tr>
</tbody>
</table>

### E Preprocessor | 790

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.1</td>
<td>Introduction</td>
<td>791</td>
</tr>
<tr>
<td>E.2</td>
<td>#include Preprocessing Directive</td>
<td>791</td>
</tr>
<tr>
<td>E.3</td>
<td>#define Preprocessing Directive: Symbolic Constants</td>
<td>792</td>
</tr>
<tr>
<td>E.4</td>
<td>#define Preprocessing Directive: Macros</td>
<td>792</td>
</tr>
<tr>
<td>E.5</td>
<td>Conditional Compilation</td>
<td>794</td>
</tr>
<tr>
<td>E.6</td>
<td>#error and #pragma Preprocessing Directives</td>
<td>795</td>
</tr>
<tr>
<td>E.7</td>
<td>Operators # and ##</td>
<td>796</td>
</tr>
<tr>
<td>E.8</td>
<td>Predefined Symbolic Constants</td>
<td>796</td>
</tr>
<tr>
<td>E.9</td>
<td>Assertions</td>
<td>797</td>
</tr>
<tr>
<td>E.10</td>
<td>Wrap-Up</td>
<td>797</td>
</tr>
</tbody>
</table>

### Index | 799

### Online Chapters and Appendices

Chapter 24 and Appendices F–K are PDF documents posted online at www.informit.com/title/9780133439854

### 24 C++11 Additional Features | 24-1
Contents

F  C Legacy Code Topics  F-1
G  UML 2: Additional Diagram Types  G-1
H  Using the Visual Studio Debugger  H-1
I  Using the GNU C++ Debugger  I-1
J  Using the Xcode Debugger  J-1
K  Test Driving a C++ Program on Mac OS X  K-1

[Note: The test drives for Windows and Linux are in Chapter 1.]
Welcome to C++11 for Programmers! This book presents leading-edge computing technologies for software developers.

We focus on software engineering best practices. At the heart of the book is the Deitel signature “live-code approach”—concepts are presented in the context of complete working programs, rather than in code snippets. Each complete code example is accompanied by live sample executions. All the source code is available at

www.deitel.com/books/cpp11fp

As you read the book, if you have questions, we’re easy to reach at
deitel@deitel.com

We’ll respond promptly. For book updates, visit www.deitel.com/books/cpp11fp. Join our social media communities on Facebook (www.deitel.com/DeitelFan), Twitter (@deitel), Google+ (gplus.to/deitel) and LinkedIn (bit.ly/DeitelLinkedIn), and subscribe to the Deitel® Buzz Online newsletter (www.deitel.com/newsletter/subscribe.html).

Features

Here are the key features of C++11 for Programmers.

C++11 Standard

The new C++11 standard, published in 2011, motivated us to write C++11 for Programmers. Throughout the book, each new C++11 feature we discuss is marked with the “11” icon you see here in the margin. These are some of the key C++11 features of this new edition:

- **Conforms to the new C++11 standard.** Extensive coverage of many of the key new C++11 features (Fig. 1).
- **Code thoroughly tested on three popular industrial-strength C++11 compilers.** We tested the code examples on GNU™ C++ 4.7, Microsoft® Visual C++® 2012 and Apple® LLVM in Xcode® 4.5.
- **Smart pointers.** Smart pointers help you avoid dynamic memory management errors by providing additional functionality beyond that of built-in pointers. We discuss unique_ptr in Chapter 17, and shared_ptr and weak_ptr in Chapter 24.
• **Earlier coverage of template-based Standard Library containers, iterators and algorithms, enhanced with C++11 capabilities.** We moved the treatment of Standard Library containers, iterators and algorithms from Chapter 20 in the previous edition to Chapters 15 and 16 and enhanced it with new C++11 features. The vast majority of your data structure needs can be fulfilled by reusing these Standard Library capabilities.

• **Online Chapter 24, C++11: Additional Topics.** In this chapter, we present additional C++11 topics. The new C++11 standard has been available since 2011, but not all C++ compilers have fully implemented the features. If all three of our key compilers already implemented a particular C++11 feature at the time we wrote this book, we generally integrated a discussion of that feature into the text with a live-code example. If any of these compilers had not implemented that feature, we included a bold italic heading followed by a brief discussion of the feature. Many of those discussions will be expanded in online Chapter 24 as the features are implemented. Placing the chapter online allows us to evolve it dynamically. This
chapter includes discussions of regular expressions, the shared_ptr and weak_ptr smart pointers, move semantics and more. You can access this chapter at:

www.informit.com/title/9780133439854

- **Random Number generation, simulation and game playing.** To help make programs more secure (see Secure C++ Programming on the next page), we now discuss C++11’s new non-deterministic random-number generation capabilities.

**Object-Oriented Programming**

- **Early-objects approach.** The book introduces the basic concepts and terminology of object technology in Chapter 1. You’ll develop your first customized C++ classes and objects in Chapter 3.

- **C++ Standard Library string.** C++ offers two types of strings—string class objects (which we begin using in Chapter 3) and C strings (from the C programming language). We’ve replaced most occurrences of C strings with instances of C++ class string to make programs more robust and eliminate many of the security problems of C strings. We discuss C strings later in the book to prepare you for working with the legacy code in industry. In new development, you should favor string objects.

- **C++ Standard Library array.** Our primary treatment of arrays now uses the Standard Library’s array class template instead of built-in, C-style, pointer-based arrays. We also cover built-in arrays because they still have some uses in C++ and so that you’ll be able to read legacy code. C++ offers three types of arrays—class templates array and vector (which we start using in Chapter 7) and C-style, pointer-based arrays which we discuss in Chapter 8. As appropriate, we use class template array and occasionally, class template vector, instead of C arrays throughout the book. In new development, you should favor class templates array and vector.

- **Crafting valuable classes.** A key goal of this book is to prepare you to build valuable reusable C++ classes. In the Chapter 10 case study, you’ll build your own custom Array class. Chapter 10 begins with a test-drive of class template string so you can see an elegant use of operator overloading before you implement your own customized class with overloaded operators.

- **Case studies in object-oriented programming.** We provide case studies that span multiple sections and chapters and cover the software development lifecycle. These include the GradeBook class in Chapters 3–7, the Time class in Chapter 9 and the Employee class in Chapters 11–12. Chapter 12 contains a detailed diagram and explanation of how C++ can implement polymorphism, virtual functions and dynamic binding “under the hood.”

- **Optional case study: Using the UML to develop an object-oriented design and C++ implementation of an ATM.** The UML™ (Unified Modeling Language™) is the industry-standard graphical language for modeling object-oriented systems. We introduce the UML in the early chapters. Chapters 22 and 23 include an optional case study on object-oriented design using the UML. We design and implement the software for a simple automated teller machine (ATM). We analyze a typical requirements document that specifies the system to be built. We determine the classes
needed to implement that system, the attributes the classes need to have, the behaviors the classes need to exhibit and we specify how the classes must interact with one another to meet the system requirements. From the design we produce a complete C++ implementation. Readers often report that the case study “ties it all together” and helps them achieve a deeper understanding of object orientation.

• **Exception handling.** We integrate basic exception handling early in the book. You can easily pull more detailed material forward from Chapter 17, Exception Handling: A Deeper Look.

• **Key programming paradigms.** We discuss *object-oriented programming* and *generic programming.*

**Pedagogic Features**

• **Examples.** We include a broad range of example programs selected from computer science, business, simulation, game playing and other topics.

• **Illustrations and figures.** Abundant tables, line drawings, UML diagrams, programs and program outputs are included.

**Other Features**

• **Pointers.** We provide thorough coverage of the built-in pointer capabilities and the intimate relationship among built-in pointers, C strings and built-in arrays.

• **Debugger appendices.** We provide three debugger appendices—Appendix H, Using the Visual Studio Debugger, Appendix I, Using the GNU C++ Debugger and Appendix J, Using the Xcode Debugger.

**Secure C++ Programming**

It’s difficult to build industrial-strength systems that stand up to attacks from viruses, worms, and other forms of “malware.” Today, via the Internet, such attacks can be instantaneous and global in scope. Building security into software from the beginning of the development cycle can greatly reduce vulnerabilities.

The CERT® Coordination Center ([www.cert.org](http://www.cert.org)) was created to analyze and respond promptly to attacks. CERT—the Computer Emergency Response Team—is a government-funded organization within the Carnegie Mellon University Software Engineering Institute™. CERT publishes and promotes secure coding standards for various popular programming languages to help software developers implement industrial-strength systems that avoid the programming practices that leave systems open to attacks.

We’d like to thank Robert C. Seacord, Secure Coding Manager at CERT and an adjunct professor in the Carnegie Mellon University School of Computer Science. Mr. Seacord was a technical reviewer for our book, *C How to Program, 7/e*, where he scrutinized our C programs from a security standpoint, recommending that we adhere to the CERT C Secure Coding Standard.

We’ve done the same for *C++11 for Programmers*, adhering to key CERT C++ Secure Coding Standard guidelines (as appropriate for a book at this level), which you can find at: [www.securecoding.cert.org](http://www.securecoding.cert.org)
We were pleased to discover that we’ve already been recommending many of these coding practices in our books since the early 1990s. If you’ll be building industrial-strength C++ systems, *Secure Coding in C and C++, Second Edition* (Robert Seacord, Addison-Wesley Professional) is a must read.

**Training Approach**

*C++11 for Programmers* stresses program clarity and concentrates on building well-engineered software.

**Live-Code Approach.** The book includes hundreds of “live-code” examples—each new concept is presented in the context of a complete working C++ program that is immediately followed by one or more actual executions showing the program’s inputs and outputs.

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

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

**Code Highlighting.** We place light-gray rectangles around each program’s key code segments.

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

**Web Access.** All of the source-code examples can be downloaded from:

www.deitel.com/books/cpp11fp

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

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

**Good Programming Practice**

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

**Common Programming Error**

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

**Error-Prevention Tip**

These tips contain suggestions for exposing and removing bugs from your programs; many of the tips describe aspects of C++ that prevent bugs from getting into your programs.
Online Chapter and Appendices

The following chapter and appendices are available online:

- Chapter 24, C++11: Additional Features
- Appendix F, C Legacy Code Topics
- Appendix G, UML 2: Additional Diagram Types
- Appendix H, Using the Visual Studio Debugger
- Appendix I, Using the GNU C++ Debugger
- Appendix J, Using the Xcode Debugger
- Appendix K, Test Driving a C++ Program on Mac OS X

To access the online chapter and appendices, go to:

www.informit.com/register

You must register for an an InformIT account and then login. After you’ve logged into your account, you’ll see the Register a Product box. Enter the book’s ISBN (9780133439854) to access the page with the online chapter and appendices.

Obtaining the Software Used in C++11 for Programmers

We wrote the code examples in C++11 for Programmers using the following C++ development tools:

- Microsoft’s free Visual Studio Express 2012 for Windows Desktop, which includes Visual C++ and other Microsoft development tools. This runs on Windows 7 and 8 and is available for download at

  www.microsoft.com/express

- GNU’s free GNU C++ (gcc.gnu.org/install/binaries.html), which is already installed on most Linux systems and can also be installed on Mac OS X and Windows systems.

- Apple’s free Xcode, which OS X users can download from the Mac App Store.
C++11 Fundamentals: Parts I, II, III and IV LiveLessons Video Training Product

Our C++11 Fundamentals: Parts I, II, III and IV LiveLessons video training product shows you what you need to know to start building robust, powerful software with C++. It includes 20+ hours of expert training synchronized with C++11 for Programmers. For additional information about Deitel LiveLessons video products, visit www.deitel.com/livelessons or contact us at deitel@deitel.com. You can also access our LiveLessons videos if you have a subscription to Safari Books Online (www.safaribooksonline.com). These LiveLessons will be available in the Summer of 2013.

Acknowledgments

We’d like to thank Abbey Deitel and Barbara Deitel of Deitel & Associates, Inc. for long hours devoted to this project. Abbey co-authored Chapter 1 and this Preface, and she and Barbara painstakingly researched the new capabilities of C++11.

We’re fortunate to have worked on this project with the dedicated publishing professionals at Prentice Hall/ Pearson. We appreciate the extraordinary efforts and mentorship of our friend and professional colleague Mark L. Taub, Editor-in-Chief of Pearson Technology Group. Carole Snyder did a great job recruiting distinguished members of the C++ community to review the manuscript. Chuti Prasertsith designed the cover with creativity and precision—we gave him our vision for the cover and he made it happen. John Fuller does a superb job managing the production of all of our Deitel Developer Series books and LiveLessons video products.

Reviewers

We wish to acknowledge the efforts of the reviewers whose constructive criticisms helped us shape the recent editions of this content. They scrutinized the text and the programs and provided countless suggestions for improving the presentation: Dean Michael Berris (Google, Member ISO C++ Committee), Danny Kalev (C++ expert, certified system analyst and former member of the C++ Standards Committee), Linda M. Krause (Elmhurst College), James P. McNellis (Microsoft Corporation), Robert C. Seacord (Secure Coding Manager at SEI/CERT, author of Secure Coding in C and C++); José Antonio González Seco (Parliament of Andalusia), Virginia Bailey (Jackson State University), Thomas J. Borrelli (Rochester Institute of Technology), Ed Brey (Kohler Co.), Chris Cox (Adobe Systems), Gregory Dai (eBay), Peter J. DePasquale (The College of New Jersey), John Dibling (SpryWare), Susan Gauch (University of Arkansas), Doug Gregor (Apple, Inc.), Jack Hagemeister (Washington State University), Williams M. Higdon (University of Indiana), Anne B. Horton (Lockheed Martin), Terrell Hull (Logica.is Integration Solutions), Ed James-Beckham (Borland), Wing-Ning Li (University of Arkansas), Dean Mathias (Utah State University), Robert A. McLain (Tidewater Community College), Robert Myers (Florida State University), Gavin Osborne (Saskatchewan Institute of Applied Science and Technology), Amar Raheja (California State Polytechnic University, Pomona), April Reagan (Microsoft), Raymond Stephenson (Microsoft), Dave Topham (Ohlone College), Anthony Williams (author and C++ Standards Committee member) and Chad Willwerth (University Washington, Tacoma).
As you read the book, we’d sincerely appreciate your comments, criticisms and suggestions for improving the text. Please address all correspondence to:

deitel@deitel.com

We’ll respond promptly. We enjoyed writing *C++11 for Programmers*. We hope you enjoy reading it!

*Paul Deitel*

*Harvey Deitel*

### About the Authors

**Paul Deitel**, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT, where he studied Information Technology. Through Deitel & Associates, Inc., he has delivered hundreds of programming courses to industry, government and military clients, including Cisco, IBM, Siemens, Sun Microsystems, Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey M. Deitel, are the world’s best-selling programming-language textbook/professional book/video authors.

**Dr. Harvey Deitel**, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has more than 50 years of experience in computing. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University. In the 1960s, through Advanced Computer Techniques and Computer Usage Corporation, he worked on the teams building various IBM operating systems. In the 1970s, he built commercial software systems. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., in 1991 with his son, Paul Deitel. The Deitels’ publications have earned international recognition, with translations published in Chinese, Korean, Japanese, German, Russian, Spanish, French, Polish, Italian, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to corporate, academic, government and military clients.

**Deitel® Dive-Into® Series Corporate Training**

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate training organization, specializing in computer programming languages, object technology, mobile app development and Internet and web software technology. The company’s clients include many of the world’s largest corporations, government agencies, branches of the military, and academic institutions. The company offers instructor-led training courses delivered at client sites worldwide on major programming languages and platforms, including C++, Visual C++®, C, Java™, Visual C#®, Visual Basic®, XML®, Python®, object technology, Internet and web programming, Android app development, Objective-C and iOS app development and a growing list of additional programming and software development courses.
Through its 37-year publishing partnership with Prentice Hall/Pearson, Deitel & Associates, Inc., publishes leading-edge programming professional books, college textbooks and *LiveLessons* video courses. Deitel & Associates, Inc. and the authors can be reached at:

**deitel@deitel.com**

To learn more about Deitel *Dive-Into® Series* Corporate Training curriculum, visit:

**www.deitel.com/training**

To request a proposal for worldwide on-site, instructor-led training at your organization, e-mail **deitel@deitel.com**.

Individuals wishing to purchase Deitel books and *LiveLessons* video training can do so through **www.deitel.com**. Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit

**www.informit.com/store/sales.aspx**
Introduction to C++ Programming, Input/Output and Operators

Objectives
In this chapter you’ll:

- Write simple C++ programs.
- Write input and output statements.
- Use fundamental types.
- Use arithmetic operators.
- Learn the precedence of arithmetic operators.
- Write decision-making statements.
2.1 Introduction

We now introduce C++ programming. We show how to display messages on the screen and obtain data from the user at the keyboard for processing. We explain how to perform arithmetic calculations and save their results for later use. We demonstrate decision-making by showing you how to compare two numbers, then display messages based on the comparison results.

Compiling and Running Programs
At www.deitel.com/books/cpp11fp, we’ve posted videos that demonstrate compiling and running programs in Microsoft Visual C++, GNU C++ and Xcode.

2.2 First Program in C++: Printing a Line of Text

Consider a simple program that prints a line of text (Fig. 2.1). This program illustrates several important features of the C++ language. The line numbers are not part of the source code.

```cpp
// Fig. 2.1: fig02_01.cpp
// Text-printing program.
#include <iostream> // allows program to output data to the screen

// function main begins program execution
int main()
{
    std::cout << "Welcome to C++!\n"; // display message
    return 0; // indicate that program ended successfully
} // end function main
```

Welcome to C++!

Fig. 2.1  |  Text-printing program.

Comments
Lines 1 and 2

// Fig. 2.1: fig02_01.cpp
// Text-printing program.

each begin with //, indicating that the remainder of each line is a comment. The comment Text-printing program describes the purpose of the program. A comment beginning with
2.2 First Program in C++: Printing a Line of Text

// is called a single-line comment because it terminates at the end of the current line. [Note: You also may use comments containing one or more lines enclosed in /* and */.]

#include Preprocessing Directive
Line 3

#include <iostream> // allows program to output data to the screen

is a preprocessing directive, which is a message to the C++ preprocessor (introduced in Section 1.4). Lines that begin with # are processed by the preprocessor before the program is compiled. This line notifies the preprocessor to include in the program the contents of the input/output stream header <iostream>. This header is a file containing information used by the compiler when compiling any program that outputs data to the screen or inputs data from the keyboard using C++’s stream input/output. The program in Fig. 2.1 outputs data to the screen, as we’ll soon see. We discuss headers in more detail in Chapter 6 and explain the contents of <iostream> in Chapter 13.

Common Programming Error 2.1
Forgetting to include the <iostream> header in a program that inputs data from the keyboard or outputs data to the screen causes the compiler to issue an error message.

Blank Lines and White Space
Line 4 is simply a blank line. Together, blank lines, space characters and tab characters are known as whitespace. Whitespace characters are normally ignored by the compiler.

The main Function
Line 5

// function main begins program execution

is another single-line comment indicating that program execution begins at the next line.

Line 6

int main()

is a part of every C++ program. The parentheses after main indicate that main is a program building block called a function. C++ programs typically consist of one or more functions and classes (as you’ll learn in Chapter 3). Exactly one function in every program must be named main. Figure 2.1 contains only one function. C++ programs begin executing at function main, even if main is not the first function defined in the program. The keyword int to the left of main indicates that main returns an integer value. The complete list of C++ keywords can be found in Fig. 4.2. We’ll say more about return a value when we demonstrate how to create your own functions in Section 3.3. For now, simply include the keyword int to the left of main in each of your programs.

The left brace, {, (line 7) must begin the body of every function. A corresponding right brace, }, (line 11) must end each function’s body.

An Output Statement
Line 8

std::cout << "Welcome to C++\n"; // display message
Chapter 2  Introduction to C++ Programming, Input/Output and Operators

instructs the computer to perform an action—namely, to print the characters contained between the double quotation marks. Together, the quotation marks and the characters between them are called a string, a character string or a string literal. In this book, we refer to characters between double quotation marks simply as strings. Whitespace characters in strings are not ignored by the compiler.

The entire line 8, including std::cout, the << operator, the string "Welcome to C++!\n" and the semicolon (;), is called a statement. Most C++ statements end with a semicolon, also known as the statement terminator (we’ll see some exceptions to this soon). Preprocessing directives (like #include) do not end with a semicolon. Typically, output and input in C++ are accomplished with streams of characters. Thus, when the preceding statement is executed, it sends the stream of characters Welcome to C++!\n to the standard output stream object—std::cout—which is normally “connected” to the screen.

The std Namespace
The std:: before cout is required when we use names that we’ve brought into the program by the preprocessing directive #include <iostream>. The notation std::cout specifies that we are using a name, in this case cout, that belongs to namespace std. The names cin (the standard input stream) and cerr (the standard error stream)—introduced in Chapter 1—also belong to namespace std. Namespaces are an advanced C++ feature that we discuss in depth in Chapter 21, Other Topics. For now, you should simply remember to include std:: before each mention of cout, cin and cerr in a program. This can be cumbersome—the next example introduces using declarations and the using directive, which will enable you to omit std:: before each use of a name in the std namespace.

The Stream Insertion Operator and Escape Sequences
In the context of an output statement, the << operator is referred to as the stream insertion operator. When this program executes, the value to the operator’s right, the right operand, is inserted in the output stream. Notice that the operator points in the direction of where the data goes. A string literal’s characters normally print exactly as they appear between the double quotes. However, the characters \n are not printed on the screen (Fig. 2.1). The backslash (\) is called an escape character. It indicates that a “special” character is to be output. When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an escape sequence. The escape sequence \n means newline. It causes the screen cursor to move to the beginning of the next line on the screen. Some common escape sequences are listed in Fig. 2.2.
The return Statement

Line 10

\[\text{return } 0; \ // \text{indicate that program ended successfully}\]

is one of several means we’ll use to exit a function. When the return statement is used at the end of main, as shown here, the value 0 indicates that the program has terminated successfully. The right brace, }, (line 11) indicates the end of function main. According to the C++ standard, if program execution reaches the end of main without encountering a return statement, it’s assumed that the program terminated successfully—exactly as when the last statement in main is a return statement with the value 0. For that reason, we omit the return statement at the end of main in subsequent programs.

2.3 Modifying Our First C++ Program

We now present two examples that modify the program of Fig. 2.1 to print text on one line by using multiple statements and to print text on several lines by using a single statement.

Printing a Single Line of Text with Multiple Statements

Welcome to C++! can be printed several ways. For example, Fig. 2.3 performs stream insertion in multiple statements (lines 8–9), yet produces the same output as the program of Fig. 2.1. [Note: From this point forward, we use a light gray background to highlight the key features each program introduces.] Each stream insertion resumes printing where the previous one stopped. The first stream insertion (line 8) prints Welcome followed by a space, and because this string did not end with \n, the second stream insertion (line 9) begins printing on the same line immediately following the space.

```cpp
1 // Fig. 2.3: fig02_03.cpp
2 // Printing a line of text with multiple statements.
3 #include <iostream> // allows program to output data to the screen
```

Fig. 2.3 | Printing a line of text with multiple statements. (Part I of 2.)
Printing Multiple Lines of Text with a Single Statement
A single statement can print multiple lines by using newline characters, as in line 8 of
Fig. 2.4. Each time the \n (newline) escape sequence is encountered in the output stream,
the screen cursor is positioned to the beginning of the next line. To get a blank line in your
output, place two newline characters back to back, as in line 8.

2.4 Another C++ Program: Adding Integers
Our next program obtains two integers typed by a user at the keyboard, computes the sum
of these values and outputs the result using \std::cout. Figure 2.5 shows the program and
sample inputs and outputs. In the sample execution, we highlight the user’s input in bold.
The program begins execution with function main (line 6). The left brace (line 7) begins
main’s body and the corresponding right brace (line 22) ends it.

Fig. 2.4 Printing multiple lines of text with a single statement.

Fig. 2.5 Addition program that displays the sum of two integers. (Part 1 of 2.)
2.4 Another C++ Program: Adding Integers

Variable Declarations
Lines 9–11

```
int number1 = 0; // first integer to add (initialized to 0)
int number2 = 0; // second integer to add (initialized to 0)
int sum = 0; // sum of number1 and number2 (initialized to 0)
```

are declarations. The identifiers number1, number2 and sum are the names of variables. These declarations specify that the variables number1, number2 and sum are data of type int, meaning that these variables will hold integer values. The declarations also initialize each of these variables to 0.

Error-Prevention Tip 2.1

Although it’s not always necessary to initialize every variable explicitly, doing so will help you avoid many kinds of problems.

All variables must be declared with a name and a data type before they can be used in a program. Several variables of the same type may be declared in one declaration or in multiple declarations. We could have declared all three variables in one declaration by using a comma-separated list as follows:

```
int number1 = 0, number2 = 0, sum = 0;
```

This makes the program less readable and prevents us from providing comments that describe each variable’s purpose.

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 2.5  |  Addition program that displays the sum of two integers. (Part 2 of 2.)
Fundamental Types
We’ll soon discuss the type double for specifying real numbers, and the type char for specifying character data. Real numbers are numbers with decimal points, such as 3.4, 0.0 and –11.19. A char variable may hold only a single lowercase letter, a single uppercase letter, a single digit or a single special character (e.g., $ or *). Types such as int, double and char are called fundamental types. Fundamental-type names consist of one or more keywords and therefore must appear in all lowercase letters. Appendix C contains the complete list of fundamental types.

Identifiers
A variable name (such as number1) is any valid identifier that is not a keyword. An identifier is a series of characters consisting of letters, digits and underscores (_ ) that does not begin with a digit. C++ is case sensitive—uppercase and lowercase letters are different, so a1 and A1 are different identifiers.

Placement of Variable Declarations
Declarations of variables can be placed almost anywhere in a program, but they must appear before their corresponding variables are used in the program. For example, in the program of Fig. 2.5, the declaration in line 9

```cpp
int number1 = 0; // first integer to add (initialized to 0)
```

could have been placed immediately before line 14

```cpp
std::cin >> number1; // read first integer from user into number1
```
Obtaining the First Value from the User

Line 13

std::cout << "Enter first integer: "; // prompt user for data
displays Enter first integer: followed by a space. This message is called a prompt because it directs the user to take a specific action. We like to pronounce the preceding statement as “std::cout gets the string ”Enter first integer: ”.” Line 14

std::cin >> number1; // read first integer from user into number1

uses the standard input stream object cin (of namespace std) and the stream extraction operator, >>, to obtain a value from the keyboard. Using the stream extraction operator with std::cin takes character input from the standard input stream, which is usually the keyboard. We like to pronounce the preceding statement as, “std::cin gives a value to number1” or simply “std::cin gives number1.”

When the computer executes the preceding statement, it waits for the user to enter a value for variable number1. The user responds by typing an integer (as characters), then pressing the Enter key (sometimes called the Return key) to send the characters to the computer. The computer converts the character representation of the number to an integer and assigns (i.e., copies) this number (or value) to the variable number1. Any subsequent references to number1 in this program will use this same value.

The std::cout and std::cin stream objects facilitate interaction between the user and the computer.

Users can, of course, enter invalid data from the keyboard. For example, when your program is expecting the user to enter an integer, the user could enter alphabetic characters, special symbols (like # or @) or a number with a decimal point (like 73.5), among others. In these early programs, we assume that the user enters valid data. As you progress through the book, you’ll learn various techniques for dealing with the broad range of possible data-entry problems.

Obtaining the Second Value from the User

Line 16

std::cout << "Enter second integer: "; // prompt user for data

prints Enter second integer: on the screen, prompting the user to take action. Line 17

std::cin >> number2; // read second integer from user into number2

obtains a value for variable number2 from the user.

Calculating the Sum of the Values Input by the User

The assignment statement in line 19

sum = number1 + number2; // add the numbers; store result in sum

adds the values of variables number1 and number2 and assigns the result to variable sum using the assignment operator =. We like to read this statement as, “sum gets the value of number1 + number2.” Most calculations are performed in assignment statements. The = operator and the + operator are binary operators—each has two operands. In the case of the + operator, the two operands are number1 and number2. In the case of the preceding = operator, the two operands are sum and the value of the expression number1 + number2.
Disbeginning the Result
Line 21

```cpp
std::cout << "Sum is " << sum << std::endl; // display sum; end line
```
displays the character string Sum is followed by the numerical value of variable sum followed by std::endl—a stream manipulator. The name endl is an abbreviation for "end line" and belongs to namespace std. The std::endl stream manipulator outputs a new-line, then "flushes the output buffer.” This simply means that, on some systems where outputs accumulate in the machine until there are enough to “make it worthwhile” to display them on the screen, std::endl forces any accumulated outputs to be displayed at that moment. This can be important when the outputs are prompting the user for an action, such as entering data.

The preceding statement outputs multiple values of different types. The stream insertion operator “knows” how to output each type of data. Using multiple stream insertion operators (<<) in a single statement is referred to as concatenating, chaining or cascading stream insertion operations.

Calculations can also be performed in output statements. We could have combined the statements in lines 19 and 21 into the statement

```cpp
std::cout << "Sum is " << number1 + number2 << std::endl;
```
thus eliminating the need for the variable sum.

A powerful feature of C++ is that you can create your own data types called classes (we introduce this capability in Chapter 3 and explore it in depth in Chapter 9). You can then “teach” C++ how to input and output values of these new data types using the >> and << operators (this is called operator overloading—a topic we explore in Chapter 10).

### 2.5 Arithmetic

Most programs perform arithmetic calculations. Figure 2.6 summarizes the C++ arithmetic operators. The asterisk (*) indicates multiplication and the percent sign (%) is the modulus operator that will be discussed shortly. The arithmetic operators in Fig. 2.6 are all binary operators, i.e., operators that take two operands. For example, the expression number1 + number2 contains the binary operator + and the two operands number1 and number2.

**Integer division** (i.e., where both the numerator and the denominator are integers) yields an integer quotient; for example, the expression 7 / 4 evaluates to 1 and the expression 17 / 5 evaluates to 3. *Any fractional part in integer division is truncated*—no rounding occurs.

C++ provides the modulus operator, %, that yields the remainder after integer division. The modulus operator can be used only with integer operands. The expression x % y yields the remainder after x is divided by y. Thus, 7 % 4 yields 3 and 17 % 5 yields 2. In later chapters, we discuss many interesting applications of the modulus operator, such as determining whether one number is a multiple of another (a special case of this is determining whether a number is odd or even).
### Arithmetic Expressions in Straight-Line Form

Arithmetic expressions in C++ must be entered into the computer in **straight-line form**. Thus, expressions such as “a divided by b” must be written as `a / b`, so that all constants, variables and operators appear in a straight line. The algebraic notation

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

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

### Parentheses for Grouping Subexpressions

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

### Rules of Operator Precedence

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

1. Operators in expressions contained within pairs of *parentheses* are evaluated first. Parentheses are at the highest level of precedence. In cases of nested, or embedded, parentheses, such as

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

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

2. Multiplication, division and modulus operations are applied next. If an expression contains several multiplication, division and modulus operations, operators are applied from **left to right**. Multiplication, division and modulus are on the **same** level of precedence.

3. Addition and subtraction operations are applied last. If an expression contains several addition and subtraction operations, operators are applied from **left to right**. Addition and subtraction also have the **same** level of precedence.

The rules of operator precedence define the order in which C++ applies operators. When we say that certain operators are applied from left to right, we are referring to the **associativity** of the operators. For example, the addition operators (+) in the expression

\[
a + b + c
\]

<table>
<thead>
<tr>
<th>C++ operation</th>
<th>C++ arithmetic operator</th>
<th>Algebraic expression</th>
<th>C++ expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
<td>(a + 7)</td>
<td>(f + 7)</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>(p - c)</td>
<td>(p - c)</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>(bm) or (b \cdot m)</td>
<td>(b * m)</td>
</tr>
<tr>
<td>Division</td>
<td>(/)</td>
<td>(x / y)</td>
<td>(x / y)</td>
</tr>
<tr>
<td>Modulus</td>
<td>(%)</td>
<td>(r \mod s)</td>
<td>(r % s)</td>
</tr>
</tbody>
</table>

Fig. 2.6 | Arithmetic operators.
associate from left to right, so \( a + b \) is calculated first, then \( c \) is added to that sum to determine the whole expression’s value. We’ll see that some operators associate from right to left. Figure 2.7 summarizes these rules of operator precedence. We expand this table as we introduce additional C++ operators. Appendix A contains the complete precedence chart.

<table>
<thead>
<tr>
<th>Operator(s)</th>
<th>Operation(s)</th>
<th>Order of evaluation (precedence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Parentheses</td>
<td>Evaluated first. If the parentheses are nested, such as in the expression ( a * (b + c / d + e) ), the expression in the innermost pair is evaluated first. [Caution: If you have an expression such as ((a + b) * (c - d)) in which two sets of parentheses are not nested, but appear “on the same level,” the C++ Standard does not specify the order in which these parenthesized subexpressions will be evaluated.]</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Evaluated second. If there are several, they’re evaluated left to right.</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>Evaluated last. If there are several, they’re evaluated left to right.</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.7** | Precedence of arithmetic operators.

**Sample Algebraic and C++ Expressions**
Now consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its C++ equivalent. The following is an example of an arithmetic mean (average) of five terms:

\[
\text{Algebra: } m = \frac{a + b + c + d + e}{5};
\]

\[
\text{C++: } m = (a + b + c + d + e)/5;
\]

The parentheses are required because division has higher precedence than addition. The entire quantity \((a + b + c + d + e)\) is to be divided by 5.

The following is an example of the equation of a straight line:

\[
\text{Algebra: } y = mx + b
\]

\[
\text{C++: } y = m * x + b;
\]

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

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

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

\[
\text{C++: } z = p * r \% q + w / x - y;
\]

The circled numbers indicate the order in which C++ applies the operators. The multiplication, modulus and division are evaluated first in left-to-right order (i.e., they associate from...
2.5 Arithmetic

left to right) because they have higher precedence than addition and subtraction. The addition and subtraction are applied next. These are also applied left to right. The assignment operator is applied last because its precedence is lower than that of any of the arithmetic operators.

**Evaluation of a Second-Degree Polynomial**

To develop a better understanding of the rules of operator precedence, consider the evaluation of a second-degree polynomial \( y = ax^2 + bx + c \):

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

The circled numbers indicate the order in which C++ applies the operators. There is no arithmetic operator for exponentiation in C++, so we’ve represented \( x^2 \) as \( x \times x \). In Chapter 5, we’ll discuss the standard library function `pow` (“power”) that performs exponentiation.

Suppose variables \( a, b, c \) and \( x \) in the preceding second-degree polynomial are initialized as follows: \( a = 2, b = 3, c = 7 \) and \( x = 5 \). Figure 2.8 illustrates the order in which the operators are applied and the final value of the expression.

![Fig. 2.8](image)

**Redundant Parentheses**

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

\[
y = ( a \times x \times x ) + ( b \times x ) + c;
\]
2.6 Decision Making: Equality and Relational Operators

We now introduce a simple version of C++’s `if` statement that allows a program to take alternative action based on whether a condition is true or false. If the condition is `true`, the statement in the body of the `if` statement is executed. If the condition is `false`, the body statement is not executed. We’ll see an example shortly.

Conditions in `if` statements can be formed by using the relational operators and equality operators summarized in Fig. 2.9. The relational operators all have the same level of precedence and associate left to right. The equality operators both have the same level of precedence, which is lower than that of the relational operators, and associate left to right.

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

Fig. 2.9 | Relational and equality operators.

Common Programming Error 2.2
Reversing the order of the pair of symbols in the operators `!=, >= and <=` (by writing them as `=!`, `=>` and `=<`, respectively) is normally a syntax error. In some cases, writing `=!` as `!=` will not be a syntax error, but almost certainly will be a logic error that has an effect at execution time. You’ll understand why when you learn about logical operators in Chapter 5. A fatal logic error causes a program to fail and terminate prematurely. A nonfatal logic error allows a program to continue executing, but usually produces incorrect results.

Common Programming Error 2.3
Confusing the equality operator `==` with the assignment operator `=` results in logic errors. We like to read the equality operator as “is equal to” or “double equals,” and the assignment operator as “gets” or “gets the value of” or “is assigned the value of.” As you’ll see in Section 5.9, confusing these operators may not necessarily cause an easy-to-recognize syntax error, but may cause subtle logic errors.

Using the `if` Statement
The following example (Fig. 2.10) uses six `if` statements to compare two numbers input by the user. If the condition in any of these `if` statements is satisfied, the output statement associated with that `if` statement is executed.
```cpp
// Fig. 2.13: fig02_13.cpp
// Comparing integers using if statements, relational operators
// and equality operators.
#include <iostream> // allows program to perform input and output
using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl

// function main begins program execution
int main()
{
    int number1 = 0; // first integer to compare (initialized to 0)
    int number2 = 0; // second integer to compare (initialized to 0)

    cout << "Enter two integers to compare: "; // prompt user for data
    cin >> number1 >> number2; // read two integers from user

    if ( number1 == number2 )
        cout << number1 << " == " << number2 << endl;
    if ( number1 != number2 )
        cout << number1 << " != " << number2 << endl;
    if ( number1 < number2 )
        cout << number1 << " < " << number2 << endl;
    if ( number1 > number2 )
        cout << number1 << " > " << number2 << endl;
    if ( number1 <= number2 )
        cout << number1 << " <= " << number2 << endl;
    if ( number1 >= number2 )
        cout << number1 << " >= " << number2 << endl;
} // end function main
```

**Fig. 2.10** | Comparing integers using if statements, relational operators and equality operators.
using Declarations

Lines 6–8

```cpp
using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl
```

are using declarations that eliminate the need to repeat the std:: prefix as we did in earlier programs. We can now write cout instead of std::cout, cin instead of std::cin and endl instead of std::endl, respectively, in the remainder of the program.

In place of lines 6–8, many programmers prefer to provide the using directive

```cpp
using namespace std;
```

which enables a program to use all the names in any standard C++ header (such as <iostream>) that a program might include. From this point forward in the book, we’ll use the preceding directive in our programs. In Chapter 21, Other Topics, we’ll discuss some issues with using directives in large-scale systems.

Variable Declarations and Reading the Inputs from the User

Lines 13–14

```cpp
int number1 = 0; // first integer to compare (initialized to 0)
int number2 = 0; // second integer to compare (initialized to 0)
```

declare the variables used in the program and initializes them to 0.

The program uses cascaded stream extraction operations (line 17) to input two integers. Remember that we’re allowed to write cin (instead of std::cin) because of line 7. First a value is read into variable number1, then a value is read into variable number2.

Comparing Numbers

The if statement in lines 19–20

```cpp
if ( number1 == number2 )
    cout << number1 << "==" << number2 << endl;
```

compares the values of variables number1 and number2 to test for equality. If the values are equal, the statement in line 20 displays a line of text indicating that the numbers are equal. If the conditions are true in one or more of the if statements starting in lines 22, 25, 28, 31 and 34, the corresponding body statement displays an appropriate line of text.

Each if statement in Fig. 2.10 has a single statement in its body and each body statement is indented. In Chapter 4 we show how to specify if statements with multiple-statement bodies (by enclosing the body statements in a pair of braces, { }, creating what’s called a compound statement or a block).

Common Programming Error 2.4

Placing a semicolon immediately after the right parenthesis after the condition in an if statement is often a logic error (although not a syntax error). The semicolon causes the body of the if statement to be empty, so the if statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the if statement now becomes a statement in sequence with the if statement and always executes, often causing the program to produce incorrect results.
White Space
Recall that whitespace characters, such as tabs, newlines and spaces, are normally ignored by the compiler. So, statements may be split over several lines and may be spaced according to your preferences. It’s a syntax error to split identifiers, strings (such as “hello”) and constants (such as the number 1000) over several lines.

Good Programming Practice 2.8
A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose meaningful breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines and left-align the group of indented lines.

Operator Precedence
Figure 2.11 shows the precedence and associativity of the operators introduced in this chapter. The operators are shown top to bottom in decreasing order of precedence. All these operators, with the exception of the assignment operator =, associate from left to right. Addition is left-associative, so an expression like \( x + y + z \) is evaluated as if it had been written \((x + y) + z\). The assignment operator = associates from right to left, so an expression such as \( x = y = 0 \) is evaluated as if it had been written \( x = (y = 0) \), which, as we’ll soon see, first assigns 0 to \( y \), then assigns the result of that assignment—0—to \( x \).

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>[See caution in Fig. 2.7]</td>
<td>grouping parentheses</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
<td>stream insertion/extraction</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
</tbody>
</table>

Fig. 2.11  | Precedence and associativity of the operators discussed so far.

Good Programming Practice 2.9
Refer to the operator precedence and associativity chart (Appendix A) when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you’re uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you’d do in an algebraic expression. Be sure to observe that some operators such as assignment (=) associate right to left rather than left to right.

2.7 Wrap-Up
You learned many important basic features of C++ in this chapter, including displaying data on the screen, inputting data from the keyboard and declaring variables of fundamen-
tal types. In particular, you learned to use the output stream object cout and the input stream object cin to build simple interactive programs. We explained how variables are stored in and retrieved from memory. You also learned how to use arithmetic operators to perform calculations. We discussed the order in which C++ applies operators (i.e., the rules of operator precedence), as well as the associativity of the operators. You also learned how C++’s if statement allows a program to make decisions. Finally, we introduced the equality and relational operators, which you use to form conditions in if statements.

The non-object-oriented applications presented here introduced you to basic programming concepts. As you’ll see in Chapter 3, C++ applications typically contain just a few lines of code in function main—these statements normally create the objects that perform the work of the application, then the objects “take over from there.” In Chapter 3, you’ll learn how to implement your own classes and use objects of those classes in applications.
This page intentionally left blank
airline reservation system 456
alert escape sequence (’\a’) 23, 637
algebraic expression 29
<algorithm> header 141, 492
algorithms 476, 485
accumulate 529, 532
all_of 533, 536
any_of 533, 536
binary_search 209, 533, 536
copy_backward 539
copy_n 541
count 529, 532
count_if 529, 532
equal 523
equal_range 546, 548
f11_t 520, 521
f11_l 520, 521
find 533, 535
find_if 533, 536
find_if_not 533, 537
for_each 533, 537
generate 520, 521
generate_n 520, 521
includes 544
insert_sorted 541, 542
is_heap 551
is_heap_until 551
iter_swap 537, 538
lexicographical_compare 524
lower_bound 548
make_heap 550
max 552
max_element 529, 532
merge 538, 540
min 552
min_element 529, 532
minmax 552
minmax_element 529, 532, 553
mismatch 522, 524
move 540
move_backward 540
none_of 533, 536
pop_heap 551
push_heap 551
random_shuffle 529, 531
remove 524, 526
remove_copy 526
remove_copy_if 524, 527, 541
remove_if 524, 526
replace 529
replace_copy 527, 529
replace_copy_if 527, 529
reverse 538, 541
reverse_copy 541, 542
separated_from_container 519
set_difference 543, 545
set_intersection 543, 545
set_symmetric_difference 543, 545
set_union 543, 546
sort 209, 533, 536
sort_heap 550
swap 537, 538
swap_ranges 537, 538
transform 529, 533
unique 538, 540
unique_copy 541, 542
upper_bound 548
alias 166, 167, 609
for the name of an object 276
alignment 617
all_of algorithm 533, 536, 558
allocate 321
allocate dynamic memory 575
allocate memory 140, 321
allocator 493
allocator_type 479
alphabetizing strings 643
alter the flow of control 121
ambiguity problem 672, 677
American National Standards Institute (ANSI) 2
American Standard Code for Information Interchange (ASCII) 116
analysis stage of the software life cycle 688
analyzing a project’s requirements 688
array subscript operator ([]) 28
arrays 209, 208
argument coercion 514
arg in correct order 136
argument passed to member-object constructor 283
argument for a macro 792
argument to a function 41
arguments in correct order 136
arguments passed to member-object constructor 283
arithmetic assignment operators 95, 96
arithmetic calculations 28
arithmetic operator 95
arithmetic operator precedence 124, 126
arithmetic operators 95
arithmetic overflow 84, 570
arithmetic operator precedence 124, 126
array subscript operator ([]) 328
array subscript operator [ ] 71
array subscript operator [ ] 265
array subscript operator [ ] 293
array subscript operator [ ] 720
ASCII (American Standard Code for Information Interchange) 720
Character Set 116, 252, 422
assert 797
assign member function of class 533
assign member function of class 729
assignment operators 27, 35, 95, 279, 308, 480
*= multiplication assignment operator 96
/= division assignment operator 96
%= modulus assignment operator 96
+= addition assignment operator 95
-= subtraction assignment operator 96
assignment statement 27, 97
associate from left to right 35, 98
associate from left to right 35, 98, 116
association 508
association (in the UML) 694, 695, 696, 728, 729
name 695
associative container 480, 483, 500, 503
ordered 476, 500
unordered 476, 500
associative container functions 503
equal_range 503
find 503
insert 503, 507
lower_bound 503
upper_bound 503
associativity 124, 126
associativity chart 35
associativity not changed by overloading 309
associativity of operators 29, 35
asterisk (*) 28
asynchronous call 717
asynchronous event 570
at member function 493, 514
class string 308, 595
class vector 226
ATM (automated teller machine) case study 683, 688
ATM class (ATM case study) 693, 694
695, 699, 702, 703, 708, 714, 715
716, 717, 718, 727
ATM system 689, 690, 691, 692, 693, 698, 703, 707, 726
atof 647
atoi 647
atol 648
attribute 45, 728, 729
component in a class diagram 701
declaration in the UML 701, 703
attribute (cont.)  
  in the UML 5, 41, 693, 698, 700, 701, 703, 707, 736  
  name in the UML 701  
of a class 4  
of an object 5  
attributes of a variable 152  
auto keyword 213  
auto_ptr object manages dynamically allocated memory 577  
automated teller machine 456  
automated teller machine (ATM) 683, 684, 688  
user interface 684  
automatic array 188  
automatic array initialization 199  
automatic local array 199  
automatic local variable 153, 156, 167  
americal object 571  
amtomatic storage class 571  
amtomatic storage duration 153  
amtomatically destroyed 156  
average 30  
average calculation 79, 85  
avoid naming conflicts 291  
avoid repeating code 270  

B  
back member function of queue 511  
back member function of sequence containers 492  
back_inserter function template 540, 542  
backslash (\) 22, 794  
backslash escape sequence (\) 23  
backward traversal 607  
bad member function 442  
bad_alloc 493, 572, 574, 578  
bad_cast exception 578  
bad_typeid exception 578  
badbit 449  
badbit of stream 422, 442  
BalanceInquiry class (ATM case study) 693, 697, 699, 700, 702, 704, 708, 715, 716, 717, 718, 727, 733, 734, 735  
Bank account program 466  
BankDatabase class (ATM case study) 693, 697, 699, 708, 715, 716, 717, 718, 720, 727, 729  
banking system 456  
bar chart 193, 194  
bar chart printing program 194  
bar of asterisks 193, 194  
base 2 621  
base case(s) 175, 179, 182  
base class 343, 345, 373  
base-class catch 578  
base-class constructor 370  
base-class exception 578  
base-class member accessibility in derived class 371  
base-class pointer to a derived-class object 389  
base-class private member 361  
base-class subobject 678  
based 132  
base specified for a stream 436  
base-number system 132, 432  
base-number system 432  
base-number system 342  
base-class initializer syntax 359  
base-class member function redefined in a derived class 368  
BasePlusCommissionEmployee class  
header 402  
BasePlusCommissionEmployee class  
implementation file 402  
BasePlusCommissionEmployee class  
represents an employee who receives a base salary in addition to a commission 352  
BasePlusCommissionEmployee class  
test program 355  
BasePlusCommissionEmployee class  
that inherits from class CommissionEmployee, which does not provide protected data 367  
base class 368  
basic searching and sorting algorithms of the Standard Library 533  
basic_fstream class template 420, 446  
basic_ifstream class template 420, 445  
basic_ios class template 418, 446, 477, 677  
basic_ostream class template 418, 420, 446, 677  
basic_stream class template 418, 446, 677  
basic_string class template 609  
basic_string class template 446  
basic_string class template 609  
basic_string class template 609  
basic_string class template 592  
basic_string class template 446  
basic_string class template 609  
basic_string class template 609  
basic_string class template 446  
basic_string class template 609  
basic_string class template 592  
basic_string class template 609  
binary_search algorithm 209, 533, 536, 558  
binary operator keywords 669  
binary operator keyword 669  
“bits-and-bytes” level 621  
bitset 513, 514, 515  
bitset header 140  
bitwise AND assignment 669  
bitwise AND assignment operator (&=) 629  
bitwise AND operator (&) 621, 624, 626  
bitwise AND, bitwise inclusive-OR, bitwise exclusive-OR and bitwise complement operators 624  
bitwise assignment operator keywords 669  
bitwise assignment operators 515, 629  
bitwise complement 622, 669  
bitwise complement operator (--) 621, 624, 627, 629, 788  
bitwise exclusive OR 669  
bitwise exclusive OR assignment operator (|=) 629  
bitwise exclusive OR operator (a) 621, 624, 627  
bitwise inclusive OR 669  
bitwise inclusive OR operator (a) 621, 624, 626  
bitwise left-shift operator (<<) 304, 627  
bitwise logical OR 515  
bitwise operator keywords 669  
bitwise operators 621, 622, 629  
bitwise right-shift operator (>>) 304  
bitwise shift operator 627  
BlackBerry OS 15  
block 34, 44, 66, 77, 89, 153, 155, 156  
block is active 153  
block is exited 153  
block of data 655  
block of memory 498, 655  
block scope 265  
variable 265  
body of a class definition 39  
body of a function 21, 22, 40  
body of a loop 102, 105  
Booch, Grady 684  
bool data type 74  
bool value false 74  
bool value true 74  
boolalpha stream manipulator 125, 432, 438  
Boolean attribute in the UML 699  
Boost C++ Library 17  
boundary of a storage unit 633  
bounds checking 198  
braces (( )) 22, 34, 66, 77, 89, 117  
braces in a do...while statement 112  
brace (( )) 187  
brk statement 118, 121
class hierarchy

class diagram (UML)

class scope

class template 186,
class-scope variable is hidden 265

class variable

Classes

class (cont.)

name 728

naming convention 39

object of 46

public services 58

services 49

class average on a quiz 79

class average problem 79

class definition 39

class development 324

class diagram

for the ATM system model 697, 722

in the UML 690, 693, 696, 698,

701, 708, 726, 729, 735, 736, 737

class diagram (UML) 41

class hierarchy 344, 389, 391

class-implementation programmer 62
class keyword 173, 583

class libraries 109

class library 372

class members default to private access 616

class scope 155, 261, 264

class-scope variable is hidden 265

class template 186, 582, 592

auto_ptr 575
definition 582

scope 585

specialization 582

Stack 583, 585

class variable 207

Classes 3

Array 324

bitset 477, 513, 514, 515
deque 485, 498

exception 562

forward_list 485, 494
invalid_argument 578

list 485, 494

multimap 505

out_of_range exception class 226

priority_queue 512

queue 511, 511

runtime_error 562, 570

set 504

stack 509

string 43

unique_ptr 575

vector 221

classic stream libraries 417

clear function of ios_base 442
clear member function of containers 479
clear member function of first-class containers 494

client code 376

client-code programmer 62

client of a class 707, 717

client of an object 48

<climits> header 141

cllog (standard error buffered) 418, 419, 445
close member function of ofstream 450

<cmath> header 109, 140
code 5

CodeLite 7

coin tossing 141

collaboration 714, 715, 717
collaboration diagram in the UML 691, 716
collaboration in the UML 714

colon (:) 155, 286, 675
column 211
column headings 189
column subscript 211

comma operator (,) 104, 181

coma-separated list of parameters 136

25, 35, 104, 230

command-line argument 240

comma-separated list of base classes 675

comment 20

CommissionEmployee class header 399

CommissionEmployee class implementation file 400

CommissionEmployee class represents an employee paid a percentage of gross sales 347

CommissionEmployee class test program 350

CommissionEmployee class uses member functions to manipulate its private data 365

Common Programming Errors overview xxiii

communication diagram in the UML 691, 716, 717

commutative 336

commutative operation 336

comparator function object 501, 505

comparator function object less 501, 512

compare iterators 484

compare member function of class string 597

comparing

strings 639, 642

comparing blocks of memory 655

comparing strings 595

compilation error 95

compilation unit 667

compiled into executable program 56

complex conditions 123

component 3

composition 284, 283, 287, 343, 346, 695, 696, 721

compound interest 108

calculation with for 108

compound statement 34

computing the sum of the elements of an array 193

concatenate 595

stream insertion operations 28

strings 641

concrete class 390

class derived from base class 395

condition 32, 73, 75, 111, 122

conditional compilation 791

conditional execution of preprocessing directives 791

conditional expression 75

classification operator (?:) 75

conditional preprocessing directives 794

conditionally compiled output statement 795

confusing equality (==) and assignment (=) operators 32, 127

conserving memory 153

consistent state 65

const 281, 313, 792

const keyword 163

const member function 40, 281

const member function on a const object 282

const member function on a non-const object 282

const object 192, 282

const object must be initialized 192

const objects and const member functions 282

const qualifier 191, 240, 661

const qualifier before type specifier in parameter declaration 166

const variables must be initialized 192

const version of operator ( ) 335

const with function parameters 240

const_cast cast away const-ness 662

const_cast demonstration 662

const_cast operator 661, 662, 663

const_iterator 479, 480, 481, 483, 484, 503, 505, 608

const_pointer 479

const_reference 479

const_reverse_iterator 479, 480, 484, 490, 608

current integral expression 119

constant pointer
to an integer constant 243
to constant data 241, 243, 244
to nonconstant data 241, 243

costant reference 334

constant reference parameter 166

constant variable 191

“const-ness” 663

constructed inside out 288

constructor 50

cannot be virtual 389

cannot specify a return type 50

conversion 377, 339

copy 332
default 53
default arguments 269
defining 52

explicit 339

function prototype 59

in a UML class diagram 54

inherit 370

inherit from base class 370

naming 52

parameter list 52

single argument 339, 340

constructors and destructors called automatically 272

container 140, 475, 476

container adapter 476, 477, 483, 509

priority_queue 512

queue 511

stack 509

Index 803
container adapter functions
pop 509
push 509
containers begin function 479
cbegin function 479
cend function 479
clear function 479
crbegin function 479
crend function 479
empty function 478
end function 479
erase function 479
insert function 478
max_size function 478
rbegin function 479
rend function 479
size function 478
swap function 478
continue statement 121
continue statement terminating a single iteration of a for statement 122
control characters 637
crbegin member function of containers
crend member function of containers
crbegin member function of class string 461, 607
copy member function of class string 461, 607
copy of the argument 241
copy_backward algorithm 539, 558
copy_if algorithm 558
copy_if algorithm 539, 558
copy_andpaste approach 356
copying strings 640
correct number of arguments 136
correct order of arguments 136
correctly initializing and using a constant variable 191
cos function 131
cosine 131
count algorithm 529, 532, 558
count function of associative container 503
count_if algorithm 529, 532, 558
counter 154
counter-controlled repetition 79, 88, 92, 101, 182
counter-controlled repetition with the for statement 103
counter variable 82
counting loop 102
cout (<<) (the standard output stream) 418, 419, 445
cout (the standard output stream) 9
cout (the standard output stream) 21, 24, 27
cout.put 421
cout.write 426
__cplusplus predefined symbolic constant 797
.cpp extension 6
CPU (central processing unit) 8
craps simulation 146, 147, 150
crbegin member function of containers 479
crbegin member function of class vector 490
create an object (instance) 40
create your own data types 28
CreateAndDestroy class
definition 273
member-function definitions 274
creating a random-access file 457
Creating a random-access file with 100 blank records sequentially 461
Creating a sequential file 447
creating an association 508
Credit inquiry program 453
credit processing program 458
crend member function of containers 479
crend member function of class vector 490
<cstdlib> header 141
<cstdlib> header 574
<cstdlib> header 140, 141, 646
<cstring> header 140, 640
<ctime> header 140, 146
<CRbs-d 116, 429, 449
<CRs-key 116
<CRs-e 116, 429, 449
<cstring> header 142
current position in a stream 452
cursor 22
 .cxx extension 6
Index 805

decrement operator (--
96
decrement operators 315
default access mode for class is private 47
default argument 167, 266
default arguments with constructors 266
default case 117, 118, 144
default constructor 51, 53, 267, 288, 316, 327, 332, 370, 478
provided by the compiler 53
provided by the programmer 53
default copy constructor 287
default delimiter 425
default memberwise assignment 279
default memberwise copy 332
default precision 90
default to decimal 436
default memberwise copy 332
define a member function of a class
38
define a constructor 52
define class GradeBook with a member function displayMessage, create a GradeBook object, and call its displayMessage function 38
Define class GradeBook with a member function that takes a parameter, create a GradeBook object and call its displayMessage function 42
#define NDEBUG 797
#define PI 3.14159
#define preprocessing directive 792
#define preprocessor directive 259
defining occurrence 9
definition 101
Deitel Buzz Online newsletter 18
delimiting constructor 271
delete 333, 575, 577
delete [] (dynamic array deallocation) 322
delete a record from a file 472
delete operator 321, 389
deleting dynamically allocated memory 333
delimiter 254, 644
delimiter (with default value \n') 423
limiting characters 644
Deposit class (ATM case study) 693, 696, 699, 700, 708, 715, 716, 724, 727, 733, 734
DepositSlot class (ATM case study) 693, 695, 699, 700, 708, 715, 716, 724, 727, 733, 734
<deque> header 140, 499
deque class template 485, 498
push_front function 499
shrink_to_fit function 499
dereference
a const iterator 482
a null pointer 232
a pointer 232, 235, 241
an iterator 480, 481, 484
an iterator positioned outside its container 490
dereferencing operator (*) 232
derive one class from another 264
derived class catch 578
descriptive words and phrases 699, 701
deserialized object 473
design process 5, 683, 689, 708, 713
design specification 689
destructor 272, 357, 478
called in reverse order of constructors 272
destructor in a derived class 370
direct selection statement 89
Dev C++ 7
diagnostics that aid program debugging 141
diamond inheritance 677
diamond symbol 71, 74
dice game 146
die rolling using an array instead of switch 195
difference_type 480
digit 26, 253, 781
direct base class 345
directly reference a value 229
disk 8, 9
disk drive 417
disk I/O completion 570
disk space 449, 573, 574
displacement 410
display screen 417, 419
divide by zero 9
DivideByZeroException 566
divides function object 554
division 29
do...while repetition statement 72, 110, 111
dollar amount 109
dot (.) operator 40
dot operator ( . ) 265, 293, 384, 376
dotted line 71
double 26
double data type 85, 108, 138
double-ended queue 498
double-precision floating-point number 89
double quote 22, 23
double-selection statement 72
double-words boundary 617
'double initializing' member objects 288
doubly linked list 494, 477
downcasting 383
driver program 272
dummy value 85
duplicate keys 500, 505
DVD 445
dynamic binding 384, 406, 407, 410
dynamic casting 411
dynamic data structure 229
dynamic memory 575
dynamic memory management 321
dynamic storage duration 153
dynamic_cast 413, 578
dynamically allocate array of integers 328
dynamically allocated memory 279, 280, 333, 389, 575
allocate and deallocate storage 272
dynamically allocated storage 332
dynamically determine function to execute 383

e
Eclipse 7
Eclipse Foundation 15
edit 6
eedit a program 6
eeditor 6
element of an array 186
elidered UML diagram 694
#ifdef 795
embedded parentheses 29
embedded system 15
Employee class 283
definition showing composition 285
definition with a static data member to track the number of Employee objects in memory 299
define header 394
implementation file 395
member-function definitions 286, 299
empty member function of containers 478
of priority_queue 513
of queue 511
of sequence container 493
of stack 509
of string 307, 601
empty parentheses 40, 41, 43
empty statement 78
empty string 48, 601
encapsulate 47
encapsulation 5, 49, 262, 278, 288
definition 240
end line 28
definition as member function of class string 608
definition as member function of containers 479
definition as member function of first-class container 480
definition as a sequence 536
definition as a stream 452
"end of data entry" 85
end-of-file 116, 117, 254, 442
#endif preprocessing directive 795
#endif preprocessor directive 259
declared 28, 90
declared on file 449
declared on file 449
declared on file indicator 449
declared on file key combination 449
declared on file marker 445
Enter key 27
enter key 117, 118
enum
scoped 150
specifying underlying integral type 150
unscoped 150
enum class 150
enum keyword 149
enum structure 150
enumeration 149, 792
enumeration constant 149, 794
EOF 116, 422, 425, 634
equality operators 32
equality and relational operators 33
escape sequences
escape sequence
escape early from a loop 121
escape character
error detected in a constructor 571
error bits 425
error

Exceptions 226
exception parameter 564
execution-time error
execute a program 6, 8
execution-time error 9

F
fabs function 131
Facebook 15
factorial 176, 177, 179
fall member function 442
fallbit 449
fallbit of stream 422, 426, 442
false 32
false 74, 75, 182, 438
fatal logic error 32
fatal runtime error 9
fault-tolerant programs 225, 561
Fibonacci series 179, 182
field width 110, 189, 426, 429
fields larger than values being printed 435
FIFO 477, 498
FIFO (first-in, first-out) 511
Flash memory 445, 446
flash drive 445
float data type 85, 138
floating point 432, 437
floating-point arithmetic 304
floating-point division 90
floating-point literal 89
double by default 89
floating-point number 85, 90
double data type 85
double precision 89
float data type 85
single precision 89
floating-point size limits 141
floating-point number in scientific format 437
floor function 131
flow of control 78, 88
flow of control in the if...else statement 75
flow of control in virtual function call 408
flush buffer 443
flush output buffer 28
flushing stream 426
fmod function 131
fgets function 426
fopen function 426
for each in algorithm 529, 533, 558
for in statement 102, 104
for repetition statement 102, 104
for repetition statement examples 105
for_each algorithm 529, 533, 558
force a decimal point 421
function body 40
form feed ("\f") 634, 637
formal parameter 136
formal type parameter 173
format error 442
format of floating-point numbers in scientific format 437
format state 426, 439
format-state stream manipulators 432
formatted data file processing 445
formatted I/O
formatted input/output 456
forward iterators 494
forward iterator operations 484
forward iterator 482, 518, 521, 529, 536, 538, 540
forward iterator operations 484
forward iterators 494
<forward_list> header 140
forward_list class template 476, 485, 494
splice_after member function 498
fractional parts 90
fragile software 364
free function (global function) 262
free store 321
friend function 289, 346
friend of a derived class 675
friends are not member functions 291
Friends can access private members of class 289
friendship granted, not taken 289
front member function of queue 511
front member function of sequence containers 492
front_inserter function template 540
<fstream> header 140
fstream 446, 462, 466, 471
<fstream> header 445
function 3, 9, 21, 137
argument 41
call 136
call overhead 163
call stack 159
declaration 137
definition 136, 156
empty parentheses 40, 41, 43
header 40
local variable 44
multiple parameters 44
name 154
overloading 170
parameter 41, 43
parameter list 43
prototype 58, 136, 137, 156, 165, 235
return a result 47
signature 137, 171
takes no arguments 162
trailing return type 175
function body 40
function call 41
function call operator () 340, 410
function call stack 242
function object 501, 505, 518, 553
binary 553
divides 554
equal_to 554
function object (cont.)
greater 554
greater_equal 554
less 554
less_equal 554
logical_and 554
logical_not 554
logical_or 554
minus 554
modulus 554
multiplies 554
negate 554
not_equal_to 554
plus 554
predefined in the STL 553
function object less<int> 501
function object less<T> 505, 512
function overhead 793
function overloading 416
function pointer 407, 410, 518, 792
function prototype 58, 109, 289, 792
parameter names optional 59
function prototype scope 155, 156
function scope 155, 156
function template 173, 582, 589
function template specialization 173
functional> header 141, 553
functional structure of a program 22
functions 3
functions for manipulating data in the standard library containers 141
functions with empty parameter lists 162
function-template specialization 582
fundamental type 26
fundamental types
unsigned int 82
G

game of chance 146
game of craps 147
game playing 141
gcount function of istream 426
general class average problem 85
general utilities library <cstdlib> 797
generals 376
generalization in the UML 733
generalized numeric operations 557
general-utilities library <cstdlib> 646
generate_algorithm 520, 521, 558
generate_n algorithm 520, 521, 558
generating values to be placed into elements of an array 191
generator function 520
generic algorithms 519
generic programming 582
get a value 49
g and set functions 48
get member function 422, 423
get pointer 452
getline function for use with class string 593
ggetline function of cin 424
ggetline function of the string header 43, 48
gets the value of 32
global 60
global function 130
global identifier 665
global namespace 667
global namespace scope 155, 156, 272, 298
global object constructors 272
global scope 272, 274, 667
global variable 154, 156, 158, 169, 667
golden mean 179
golden ratio 179
good function of ios_base 442
Good Programming Practices overview xxiii
goodbit of stream 442
goto elimination 70
goto statement 70
GradeBook.cpp 80, 86
GradeBook.h 80, 85
graph information 194
Graphical User Interface (GUI) 16
greater function object 554
greater_equal function object 554
greater-than operator 38
greater-than-or-equal-to operator 32
greater-than-or-equal-to operator 32
greater-than-or-equal-to operator 32
guard condition 74
guard condition in the UML 705
GUI (Graphical User Interface) 16
guillemets (‘« and ‘») in the UML 54

H
.h filename extension 54
half-word 617
handle on an object 264
hard disk 445
hardcopy printer 9
hardware platform 2
has-a relationship 343, 695, 283
header 54, 62, 139, 259, 372, 791
Headers
<algorithm> 492, 558
<cmath> 109
<deque> 499
<exception> 562
<forward_list> 494
<fstream> 445
<functional> 553
<iomani.h> 90
<iosstream> 21, 116
<list> 494
<map> 505, 507
<memory> 578
<nume> 559
<queue> 511, 512
<set> 501
<stack> 509
<stdexcept> 562, 578
<string> 43
<typeinfo> 413
<unordered_map> 505, 507
<unordered_set> 501, 504
<vector> 221
how they are located 57
name enclosed in angle brackets (< >) 57
name enclosed in quotes (" ") 57
heap 321, 512, 548, 551
heapsort sorting algorithm 548
helper function 265
hex stream manipulator 427, 432, 436

Index 807
Index 813

point notation 251
pointer operators & and * 232
pointer to a function 407
pointer to an object 242
pointer to void (void *) 249
pointer variable 575
pointer/offset notation 250
pointer/subscript notation 250
pointer-based string 606
pointers and array subscripting 249, 250
pointers and arrays 249
pointers declared const 243
pointers to dynamically allocated storage 293, 334
point-to-member operators
pointer to a function 407
pointer-to-member operators
pointers to dynamically allocated storage 293, 334
pointers and arrays 249, 250
pointer-based string 606
pointer/subscript notation 250
pointer/offset notation 250
pointer variable 575
predefined symbolic constants 796
predefined symbolic constants
predefined function objects 553
preprocessor directives 7, 21
preprocessor 6, 7, 137, 791
preprocessor directives
#define 259
#define 259
#define 259
#define 259
d#define 259
prev_permutation algorithm 559
prevent memory leak 577
primary memory 8
primitive data type promotion 90
principal 108
principal of least privilege 153, 239, 240, 243, 281, 450, 483
print a line of text 20
printer 9, 417
printing
line of text with multiple statements 23
multiple lines of text with a single statement 24
unsigned integer in bits 622
priority_queue adapter class 512
empty function 513
pop function 512
push function 512
size function 513
top function 513
private
access specifier 47, 726
base class 372
base-class data cannot be accessed from derived class 358
inheritance 345
members of a base class 345
static data member 298
private libraries 7
probability 141
program development environment 6
program execution stack 159
program in the general 375
program in the specific 375
program termination 275, 276
programmer-defined function
maximum 132
promotion 90
promotion hierarchy for built-in data types 138
promotion rules 138
prompt 27, 88
prompting message 443
proprietary classes 372
protected 361
protected access specifier 258
protected base class 372
protected base-class data can be accessed from derived class 363
protected inheritance 345, 372
pseudorandom numbers 144
public
keyword 726, 731
method 260
public access specifier 39
public base class 372
public inheritance 343, 345
public keyword 39
public member of a derived class 346
public services of a class 58
public static class member 298
public static member function 298
punctuation mark 644
pure specifier 391
pure virtual function 391, 407
push 587
push function of containers 509
push function of priority_queue 512
push member function of queue 511
push member function of stack 509
push_back member function of class template vector 227
push_back member function of vector
push_front member function of deque
push_front member function of list 497
push_heap algorithm 551, 559
put file-position pointer 457, 462
put member function 421, 422
put pointer 452
putback function of ifstream 425
Q
qualified name 369
<queue> header 140
queue adapter class 511
back function 511
empty function 511
front function 511
pop function 511
push function 511
size function 511
<queue> header 511, 512
quotation marks 22
R
radians 131
raise to a power 132
rand function 141, 142
RAND_MAX symbolic constant 141
random integers in range 1 to 6 142
random number 144
random_shuffle algorithm 529, 531, 558
random-access file 443, 457, 458, 464, 466
random-access iterator 482, 483, 498, 501, 518, 524, 531, 536, 550, 551
random-access iterator operations 482
randomizing 144
randomizing the die-rolling program 145
range 481, 532
range checking 324, 595
range-based for 595, 608
range-based for statement 200
Rational Software Corporation 690
Rational Unified Process™ 690
raw data 456
raw data processing 445
rbegin member function of container
string 608
rbegin member function of containers
479
rbegin member function of vector 490
seek put 452
seekg function of istream 452, 472
seekp function of ostream 452, 462
select a substring 340
selection statement 70, 73
self assignment 333
self-documenting 26
semicolon (;) 22, 34, 78, 791
semicolon that terminates a structure definition 617
send a message to an object 5
sentinel value 85, 89, 116
separate interface from implementation 58
sequence 210, 481, 538, 540
sequence container 476, 483, 485, 493, 497
back function 492
empty function 493
front function 492
insert function 493
sequence diagram in the UML 691, 716
sequence of messages in the UML 717
sequence of random numbers 144
sequence of messages in the UML 71, 73
sequence-statement activity diagram 71
sequential execution 70
sequential file 445, 446, 447, 450, 456
serialized object 473
services of a class 49
<set> header 140
set a value 49
set and get functions 48
set associative container 504
set function 288
<set> header 501, 504
set_intersection 545
set_new_handler function 572, 574
set of recursive calls to method Fibonacci 181
set operations of the Standard Library 543
set_difference algorithm 543, 545, 549
set_intersection algorithm 543, 545, 559
set_new_handler specifying the function to call when new fails 574
set_symmetric_difference algorithm 543, 545, 559
set_union algorithm 543, 546, 559
setbase stream manipulator 427
setfill stream manipulator 261, 433, 435
setprecision stream manipulator 90, 109, 427
setw 189, 312
setw parameterized stream manipulator 109
setw stream manipulator 254, 429, 433
Shape class hierarchy 345
shell prompt on Linux 9
shift a range of numbers 142
shifted, scaled integers 142
shifted, scaled integers produced by 1 + rand() % 6 142
shiftingValue 146
short-circuit evaluation 124
short data type 120
short int data type 120
showbase stream manipulator 432, 436
showpoint stream manipulator 91, 432
showpos stream manipulator 432, 434
shrink_to_fit member function of classes vector and deque 490
shuffle algorithm 558
shuffling algorithm 619
side effect 164
side effect of an expression 154, 164, 181
sign extension 622
sign left justified 432
sign extension 622
signature of overloaded prefix and postfix increment operators 315
significant digits 433
simple condition 122, 124
sin function 132
sine 132
single-argument constructor 339, 340
single-entry/single-exit control statement 73, 74
single inheritance 345, 677
single-line comment 21
single-precision floating-point number 89
single quote 23
single quote (') 252
single-selection if statement 72, 76
singly linked list 476, 494
six-sided die 142
size_t type 244
size_t type 492
size_t type 493
side effect 164
size of a variable 152
size of an array 244
size_t 189, 458
size_t type 244
size_t type 480
size_t 464, 651, 794
sizeof operator 244, 245, 291
used to determine standard data type sizes 245
sizeof operator when applied to an array name returns the number of bytes in the array 245
skip remainder of switch statement 121
skip remaining code in loop 122
skipping whitespace 426, 432
skipws stream manipulator 432, 434
small circle symbol 71
smart pointer 169
software engineering 58
software reuse 3, 343, 582, 672
solid circle (for representing an initial state in a UML diagram) in the UML 703, 705
solid circle enclosed in an open circle (for representing the end of a UML activity diagram) 705
solid circle symbol 71
solid diamonds (representing composition) in the UML 695
sort algorithm 209, 533, 536, 558
sort member function of list 497
sort_heap algorithm 550, 559
sorting 446, 533
sorting and related algorithms 557
sorting arrays 209
sorting strings 141
source code 6, 372
source-code file 54
SourceForge 15
spaces for padding 435
space-time trade-off 466
special character 253
special characters 26
specialization in the UML 734
spiral 179
splice member function of list 497
splice_after member function of class template forward_list 498
sort function of cmath header 132
square function 139
square root 132, 428
sqrt function 144
sqrt( time(0) ) 146
<sstream> header 140, 609, 609
stable_partition algorithm 558
stable_sort algorithm 558
<stack> header 140
stack 158, 582
stack adapter class 509
empty function 509
pop function 509
push function 509
size function 509
top function 509
Stack class template 582, 589
stack frame 159
<stack> header 509
stack overflow 159, 176
stack unwinding 566, 569, 571
Stack< double > 585, 588
stack<T> 587
standard data type sizes 245
standard error stream (cerr) 9
standard exception classes 578
standard input object (cin) 27
standard input stream (cin) 9, 418
standard input stream object (cin) 445
Standard Library class string 305
container classes 476
deque class template 499
exception classes 579
exception hierarchy 578
Index

typedef (cont.)
    iostream 418
    istream 418
    ostream 420
    ofstream 418
    typeid 413, 578
    <typeinfo> header 140, 413
typename keyword 173, 583
type-safe I/O 425

U
UML (Unified Modeling Language) 6, 41, 683, 684, 690, 694, 701, 702, 733
action expression 71
action state 71
activity diagram 70, 71, 78
arrow 71
attribute 41
class diagram 41
contactor in a class diagram 54
data types 44
decision symbol 74
diagram 690
diamond symbol 71, 74
dotted line 71
final state 71
guard condition 74
guillemets (« and ») 54
guillemets ( « and ») 71
header 140
<typeinfo> 413
<typeinfo> 578
undefined area in memory 618
underflow_error exception 579
underlying container 509
underlying data structure 512
underscore (_) 26
unformatted I/O 418, 425
unformatted output 420, 422
Unicode 592
Unicode character set 177, 178
uniform initialization 94
uniform_int_distribution 151
unincremented copy of an object 321
unique algorithm 538, 540, 558
unique keys 500, 504, 507
unique member function of list 498
unique_copy algorithm 541, 542, 558
unique_ptr class 575
built-in array 578
universal-time format 260
UNIX 449
unnamed bit field 633
unnamed bit field with a zero width 633
unnamed namespace 667
unordered associative containers 476,
    477, 500
<unordered_map> header 140
unordered_map class template 477, 507
unordered_multimap class template
    477, 505
unordered_multiset class template
    477, 501
<unordered_set> header 140
unordered_set class template 477, 504
unscoped enum 150
unsigned char data type 139
unsigned data type 139, 145
unsigned int data type 139, 145
unsigned int fundamental type 82
unsigned integer in bits 622
unsigned long 178, 650
unsigned long data type 138
unsigned long int 177, 178
unsigned long int data type 138
unsigned long long data type 138
unsigned long long int data type 138
unsigned short data type 139
unsigned short int data type 139
unint an input stream from an output
    stream 443
unwinding the function call stack 568
update records in place 456
upper_bound algorithm 548, 558
upper_bound function of associative
    container 503
uppercase letter 26, 140, 634, 636
uppercase stream manipulator 432,
    436, 438
use case diagram in the UML 689, 690
use case in the UML 688, 689

V
validation 64
validity checking 64
value 27
value initialize 238
value of a variable 152
value of an array element 187
value_type 479
variable 25
variable name 25
argument 44
parameter 44
variadic template 504
<v> header 140
vector class 221
capacity function 488, 488
crbegin function 490
crend function 490
push_back function 488
push_front function 488
rbegin function 490
rend function 490
vector class template 186, 486
push_back member function 227
shrink_to_fit member function 490
vector class template element-manipulation functions 490
<vector> header 221
verb phrase in requirements specification 707
vertical spacing 102
vertical tab (‘v’) 634, 637
vi 6
textual base class 661, 678, 679, 680,
    681
textual destructor 389
textual function 375, 383, 407, 409,
    478
call 409
call illustrated 408
table (stable) 407
textual inheritance 679
textual virtual memory 573, 574
visibility in the UML 726
visibility marker in the UML 726
Visual Studio 2012 Express Edition 7
void * 249, 655
void keyword 40, 48
void return type 138
volatile qualifier 661
volume of a cube 164
stable 407, 409, 410
stable pointer 410

while repetition statement 72, 78, 102, 110
while statement activity diagram 79
whitespace characters 21, 22, 35, 422, 423, 426, 634, 637, 791, 796
whole/part relationship 695
width implicitly set to 0 429
width member function of class ios_base 429
width of a bit field 630
width of random number range 146
width setting 429
Windows 15, 116
Windows operating system 15
Windows Phone 7 15
Withdrawal class (ATM case study) 693, 694, 695, 696, 699, 700, 705, 708, 715, 716, 719, 720, 728, 729, 731, 733, 734, 735, 737, 738, 739
word 617
word boundary 617
workflow of a portion of a software system 71
workflow of an object in the UML 704

W
“walk off” either end of an array 323
warning message 66
waterfall model 688
wchar_t 592
wchar_t character type 418
“weakest” iterator type 482, 519
what member function of an exception object 226
what virtual function of class exception 562, 567, 573
while repetition statement 72, 78, 102, 110
while statement activity diagram 79
write function of ostream 421, 425
writing data randomly to a random-access file 462

X
Xcode 7
Xerox PARC (Palo Alto Research Center) 16
XML (extensible markup language) 473
xor operator keyword 669
xor_eq operator keyword 669

Z
zeroth element 186
zero-width bit field 633