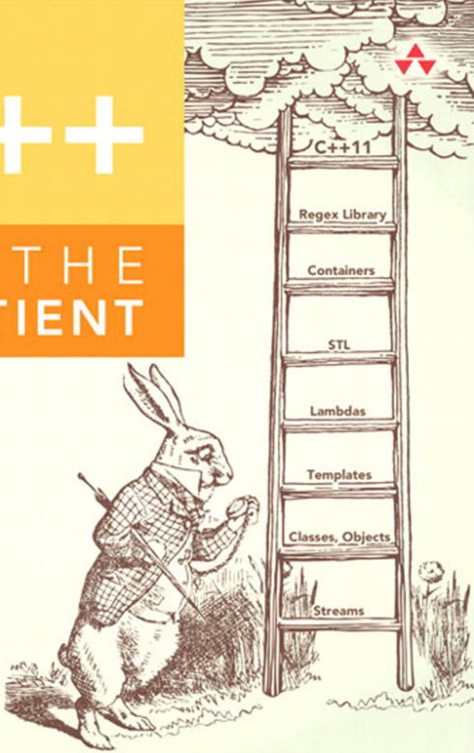
C++

FOR THE IMPATIENT



BRIAN OVERLAND

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

C++ for the Impatient

Brian Overland

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

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

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

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

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Overland, Brian, 1958–
C++ for the impatient / Brian Overland.
pages cm
Includes index.
ISBN-13: 978-0-321-88802-0 (alk. paper)
ISBN-10: 0-321-88802-2 (alk. paper)
1. C++ (Computer program language) I. Title.
QA76.73.C153O8348 2013
005.13'3--dc23
2013005331

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-88802-0
ISBN-10: 0-321-88802-2
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
First printing: May, 2013

Once more, for Colin.

This page intentionally left blank

Contents

Preface xix

Acknowledgments xxvii

About the Author xxix

1 C++ Fundamentals 1

1.1 Elements of a C++ Program 1
  1.1.1 The #include Directive 2
  1.1.2 The using Statement 2
  1.1.3 The main Function 3
1.2 Dealing with "Flashing Console" 4
1.3 Working with Microsoft Visual Studio 5
1.4 Doing More with C++ 6
1.5 Adding Simple Variable Declarations 7
1.6 Introduction to C++ Control Structures 10
  1.6.1 Making Decisions with "if" 11
  1.6.2 Looping with "while" 13
1.7 General Structure of a C++ Program 14
1.8 More about Namespaces 15
1.9 Some Comments about Comments 17
  1.9.1 C++ Comments (Line Comments) 17
  1.9.2 C-Language-Style Comments 17
1.10 Sample App: Adding Machine 19
Exercises 20
1.11 Sample App: Calculating Phi 20
Exercises 23

2 Data 25

2.1 Declaring Simple Variables 25

2.2 Primitive Data Types 27

2.2.1 Integer Types: Guidelines 28

2.2.2 Floating-Point Types: Guidelines 29

2.3 Symbolic Names ("Symbols") 30

2.4 Numeric Literals 31

2.5 Mixing Numeric Types 33

2.5.1 Integer versus Floating Point 34

2.5.2 bool versus Integer Types 34

2.5.3 Signed versus Unsigned Integers 35

2.6 String and Character Literals 39

2.6.1 Single-Quoted Characters 39

2.6.2 Double-Quoted Strings 40

2.6.3 Special Characters (Escape Sequences) 41

2.6.4 Wide-Character Strings 45

2.6.5 Raw String Literals (C++11) 46

2.7 Data Declarations: The Complete Syntax 46

2.8 Enumerated Types 50

2.9 Special Declarations (typedef, auto, decltype) 52

2.9.1 The typedef Keyword 52

2.9.2 The auto and decltype Keywords (C++11) 53

2.10 Sample App: Type Promotion 54

Exercises 55

3 Operators 57

3.1 Precedence, Associativity, and Lvalues 57

3.1.1 Precedence 58

3.1.2 Associativity 58

3.1.3 Lvalues 59

3.2 Concise Summary of Operators 59

3.3 Operators in Detail 62

3.3.1 The Scope Operator (::) 62

3.3.2 Data-Access and Other High-Precedence Operators 62
3.3.3 Prefix Operators 64
3.3.4 Pointer-to-Member Operators 67
3.3.5 Multiply and Divide 68
3.3.6 Add and Subtract 69
3.3.7 Shift Operators 69
3.3.8 Less Than and Greater Than 70
3.3.9 Test for Equality 71
3.3.10 Bitwise and Logical Conjunctions 72
3.3.11 The Conditional Operator (?:) 73
3.3.12 Assignment Operators 73
3.3.13 The throw Operator 76
3.3.14 The Join Operator (,) 76
3.4 The Great Controversy: Postfix or Prefix? 77
3.5 Bitwise Operators in Detail 78
3.5.1 Bitwise AND, OR, and Exclusive OR 78
3.5.2 Operating on Signed and Unsigned 79
3.5.3 Bitwise Shifts 81
3.6 Cast Operators 82
3.6.1 The static_cast Operator 83
3.6.2 The reinterpret_cast Operator 84
3.6.3 The const_cast Operator 85
3.6.4 The dynamic_cast Operator 86
3.6.5 The C-Language Cast 87
3.7 Sample App: Binary Printout 88
Exercises 90

4 Control Structures 91
4.1 Concise Summary of C++ Statements 91
4.2 Null Statements (;) and Expression Statements 93
4.3 Compound Statements 94
4.4 if and if-else Statements 96

4.5 while and do-while Statements ... 98
4.6 for Statements ... 99
4.7 Range-based for Statements (C++11) ... 101
4.8 switch Statements ... 103
4.9 Jump Statements (break, continue, goto) ... 104
4.10 Exception Handling (try, catch) ... 106
  4.10.1 What Is an Exception? ... 107
  4.10.2 try-catch Blocks: General Syntax ... 108
  4.10.3 catch Blocks and Exception Objects ... 109
  4.10.4 Throwing an Exception ... 110
4.11 Sample App: Guess-the-Number Game ... 111
Exercises ... 113
4.12 Sample App: Computer Guesses the Number ... 113
Exercises ... 115

5 Functions ... 117
5.1 Overview of Traditional (Named) Functions ... 117
  5.1.1 Function Prototypes versus Definitions ... 117
  5.1.2 Prototyping a Function (Simplified Syntax) ... 119
  5.1.3 Defining a Function ... 120
  5.1.4 Calling a Function ... 121
5.2 Local and Global Variables ... 122
5.3 Complete Function Declaration Syntax ... 124
5.4 Function Overloading ... 126
5.5 Arguments with Default Values ... 128
5.6 Variable-Length Argument Lists ... 129
5.7 Lambda, or Anonymous, Functions (C++11) ... 131
  5.7.1 Basic Lambda Syntax ... 132
  5.7.2 Lambda Closure Syntax ... 133
  5.7.3 The mutable Keyword ... 136
  5.7.4 Using Lambdas with the STL ... 137
  5.7.5 Storing and Returning Lambdas ... 138

5.8 constexpr Functions (C++11) 141
5.9 Sample App: Odds at Dice 142
Exercises 145

6 Pointers, Arrays, and References 147

6.1 References 147
6.1.1 Reference Arguments 147
6.1.2 Returning a Reference from a Function 150
6.1.3 References Modified by “const” 151
6.2 Arrays 152
6.2.1 Simple (One-Dimensional) Arrays 153
6.2.2 Array Processing with Loops 154
6.2.3 Passing Arrays to Functions 155
6.2.4 Multidimensional Arrays 157
6.3 Pointers 159
6.3.1 The Concept of Pointer 159
6.3.2 Pointers as Arguments 162
6.3.3 Pointers Used to Access Arrays 163
6.3.4 Pointer Arithmetic 166
6.3.5 Pointers versus Array Arguments 167
6.3.6 Pointers and Memory Allocation 167
6.3.7 Pointers to const Types 170
6.3.8 Applying const to the Pointer Itself 172
6.3.9 Void Pointers (void*) 173
6.4 Complex Declarations Involving Pointers 175
6.5 Passing and Returning Function Pointers 178
6.6 Smart Pointers (C++11) 180
6.6.1 The shared_ptr Template 181
6.6.2 The unique_ptr Template 184
6.7 Sample App: Sieve of Eratosthenes 186
Exercises 188

7 **Classes and Objects** 189
7.1 Overview: Structures, Unions, and Classes 189
7.2 Basic Class Declaration Syntax 191
7.2.1 Declaring and Using a Class 191
7.2.2 Data-Member Access (Public, Private, Protected) 193
7.2.3 Defining Member Functions 195
7.2.4 Calling Member Functions 198
7.2.5 Private Member Functions 200
7.2.6 Classes Containing Classes 201
7.2.7 Static Members 203
7.3 Constructors 205
7.3.1 The Default Constructor 206
7.3.2 Overloaded Constructors and Conversion Functions 208
7.3.3 Copy Constructor 210
7.3.4 Constructor Initialization Lists 211
7.3.5 Delegated Constructors (C++11) 214
7.3.6 Default Member Initialization (C++11) 215
7.4 Destructors 216
7.5 The Hidden "this" Pointer 217
7.6 Operator Functions (Op Overloading) 218
7.6.1 Operator Functions as Members 219
7.6.2 Operator Functions as Friends 221
7.6.3 Assignment Operator (=) 222
7.6.4 Function-Call Operator, ( ) 224
7.6.5 Subscript Operator, [ ] 225
7.6.6 Increment and Decrement (++, --) 227
7.6.7 Incoming and Outgoing Conversion Functions 228
7.7 Deriving Classes (Subclassing) 229
7.7.1 Subclass Syntax 229
7.7.2 Base-Class Access Specifiers 231
7.7.3 Inherited Constructors (C++11) 232
7.7.4 Up-casting: Child Objects and Base-Class Pointers 233

7.7.5 Virtual Functions and Overriding ... 235
7.7.6 Pure Virtual Functions ... 238
7.7.7 Override Keyword (C++11) ... 238
7.7.8 Resolving Name Conflicts within Hierarchies ... 239
7.8 Bit Fields ... 240
7.9 Unions ... 242
7.9.1 Named Unions ... 242
7.9.2 Anonymous Unions ... 244
7.10 Sample App: Packed Boolean ... 245
Exercises ... 248

8 Preprocessor Directives ... 249
8.1 General Syntax of Preprocessor Directives ... 249
8.2 Summary of Preprocessor Directives ... 250
8.3 Using Directives to Solve Specific Problems ... 254
8.3.1 Creating Meaningful Symbols with #define ... 254
8.3.2 Creating Macros with #define ... 256
8.3.3 Conditional Compilation (#if, #endif, and so on) ... 257
8.4 Preprocessor Operators ... 259
8.5 Predefined Macros ... 260
8.6 Creating Project Header Files ... 263

9 Creating and Using Templates ... 265
9.1 Templates: Syntax and Overview ... 265
9.2 Function Templates ... 267
9.2.1 Function Templates with One Parameter ... 267
9.2.2 Dealing with Type Ambiguities ... 269
9.2.3 A Function Template with Two Parameters ... 270
9.3 Class Templates ... 272
9.3.1 Simple Class Templates ... 273
9.3.2 Instantiating Class Templates ... 274

9.4 Class Templates with Member Functions 276
  9.4.1 Class Templates with Inline Member Functions 276
  9.4.2 Class Templates with Separate Function Definitions 276
9.5 Using Integer Template Parameters 278
9.6 Template Specialization 279
9.7 Variadic Templates (C++11) 281
  9.7.1 A More Sophisticated Variadic Template 284
  9.7.2 Summary of Rules for Variadic Templates 285
  9.7.3 Tuples 286
9.8 Sample App: Type Promotion, v 2 288
Exercises 289

10 C-String Library Functions 291
10.1 Overview of the C-String Format 291
10.2 Input and Output with C-Strings 293
10.3 C-String Functions 294
10.4 String Tokenizing with strtok 300
10.5 Individual-Character Functions 301
10.6 Memory-Block Functions (memcpy, and so on) 304
10.7 Wide-Character Functions (wstrcpy, and so on) 306

11 C I/O Library Functions 309
11.1 Overview of C Library I/O 309
11.2 Console I/O Functions 310
11.3 Print/Scan Formats 313
  11.3.1 printf Format Specifiers (%) 313
  11.3.2 Advanced printf Format Syntax 315
  11.3.3 Using Format Specifiers with scanf 317
11.4 Input and Output to Strings 321
11.5 File I/O 321
  11.5.1 Opening a File 322
  11.5.2 Closing a File 324

11.5.3 Reading and Writing Text Files 325
11.5.4 Reading and Writing Binary Files 327
11.5.5 Random-Access Functions 328
11.5.6 Other File-Management Functions 330

12 Math, Time, and Other Library Functions 333
12.1 Trigonometric Functions 333
12.2 Other Math Functions 336
12.3 The C Date and Time Library 339
12.3.1 Date and Time Functions 340
12.3.2 The TM (“Time”) Data Structure 342
12.3.3 Date/Time Format Specifiers (strftime) 344
12.4 String-to-Number Conversions 347
12.5 Memory-Allocation Functions 348
12.6 Standard C Randomization Functions 350
12.7 Searching and Sorting Functions 351
12.7.1 The bsearch Function (Binary Search) 352
12.7.2 The qsort Function (Quick Sort) 353
12.7.3 Comparison Functions Using C-Strings 355
12.8 Other Standard C Library Functions 355
12.9 Sample App: Idiot Savant 358
Exercises 359

13 C++ I/O Stream Classes 361
13.1 The Basics of C++ I/O Streams 361
13.1.1 Writing Output with << 362
13.1.2 Reading Input with >> 363
13.2 Reading a Line of Input with getline 364
13.3 The C++ Stream-Class Hierarchy 366
13.4 Stream Objects: Manipulators and Flags 368
13.4.1 Stream Manipulators 369
13.4.2 Stream Format Flags 377

13.5 Stream Member Functions (General Purpose) 379
  13.5.1 Input Stream Functions 379
  13.5.2 Output Stream Functions 381
  13.5.3 Flag-Setting Stream Functions 382
13.6 File Stream Operations 385
  13.6.1 Overview: Text versus Binary File I/O 385
  13.6.2 Creating a File Object 387
  13.6.3 File-Specific Member Functions 390
  13.6.4 Reading and Writing in Binary Mode 391
  13.6.5 Random-Access Operations 392
13.7 Reading and Writing String Streams 395
13.8 Overloading Shift Operators for Your Classes 398
13.9 Sample App: Text File Reader 400
Exercises 401

14 The C++ STL String Class 403
14.1 Overview of the String Class 403
14.2 String Class Constructors 405
14.3 String Class Operators 406
  14.3.1 Assigning Strings (=) 406
  14.3.2 Joining Strings (+) 407
  14.3.3 Comparing Strings 408
  14.3.4 Indexing Strings ([ ]) 408
14.4 Concise Summary of Member Functions 410
14.5 Member Functions in Detail 410
14.6 String Class Iterators 424
  14.6.1 Standard (Forward) Iterators 424
  14.6.2 Reverse Iterators 426
  14.6.3 Iterator Arithmetic 427
  14.6.4 Member Functions Using Iterators 428
14.7 Wide-Character String Class (basic_string) 430

15 Introduction to STL (vector, deque) 431

15.1 A Tour of the Container Templates 431

15.2 Introduction to Iterators 433

15.3 The vector Template 434

15.3.1 Vector Iterators 436

15.3.2 Vector Constructors 438

15.3.3 List Initialization (C++11) 439

15.3.4 Concise Summary of Vector Functions 439

15.3.5 Vector Member Functions in Detail 440

15.4 The deque Template 447

15.4.1 Deque Iterators 449

15.4.2 Deque Constructors 450

15.4.3 Concise Summary of Deque Functions 451

15.4.4 Deque Functions in Detail 452

15.5 The bitset Template 458

15.5.1 bitset Constructors 459

15.5.2 bitset Member Functions 460

15.6 Sample App: Alpha File Organizer 461

Exercises 463

16 STL Sequence Containers (List) 465

16.1 Sorting Elements (Strict Weak Ordering) 465

16.2 The list Template 466

16.2.1 List Iterators 467

16.2.2 List Constructors 469

16.2.3 Concise Summary of list Functions 471

16.2.4 List Member Functions in Detail 472

16.3 The stack Template 481

16.4 The queue Template 484

16.5 The priority_queue Template 487

16.5.1 Priority Queue Constructors 489

16.5.2 Priority Queue Member Functions 490

16.6 Sample App: Find the Median 491
Exercises 493

17 **STL Associated Containers (map, set)** 495
17.1 The pair Template 495
17.2 The map Template 497
17.2.1 Populating a Map 498
17.2.2 Finding an Existing Map Element 501
17.2.3 A More Sophisticated Record Type 502
17.2.4 Iterating through a Map 506
17.2.5 Map Implementation: Binary Trees 507
17.2.6 Map Constructors 509
17.2.7 Concise Summary of Map Functions 511
17.2.8 Map Member Functions in Detail 512
17.3 The set Template 518
17.3.1 Populating a Set 519
17.3.2 Finding an Element in a Set 520
17.3.3 Set Constructors 521
17.3.4 Concise Summary of Set Functions 523
17.3.5 Set Member Functions in Detail 524
17.4 The multimap Template 529
17.5 The multiset Template 532
17.6 Unordered Containers (C++11) 534
17.6.1 Unordered Containers: Basic Concepts 535
17.6.2 Fine-Tuning Hash-Table Performance 538
17.6.3 Writing Your Own Hash and Equals Functions 541
17.7 Sample App: Guess-the-Word Game 543
Exercises 545

18 **STL Algorithms** 547
18.1 STL Algorithms: General Concepts 547
18.2 Using Lambda Functions (C++11) 550

18.3 Algorithms and Iterators 551
18.4 Insert Iterators 553
18.5 Sample App: Finding the Median 555
18.6 Concise Summaries of Algorithms 556
  18.6.1 Read-Only Algorithms 556
  18.6.2 Modifying Algorithms 556
  18.6.3 Sorting and Reordering Algorithms 558
  18.6.4 Heap Algorithms 560
  18.6.5 Numeric Algorithms 562
  18.6.6 Predefined Function Objects 562
18.7 Detailed Descriptions of Algorithms 564

19 C++11 Randomization Library 599
19.1 Issues in Randomization 599
  19.1.1 The Problem of Biased Distribution 599
  19.1.2 The Problem of Pseudorandom Sequences 600
  19.1.3 The Problem of Getting a Seed 600
19.2 A Better Randomization Scheme 601
19.3 Common Engines 604
19.4 Common Distributions 605
19.5 Operations on Engines 608
19.6 Operations on Distributions 609
19.7 Sample App: Dice Game 610
Exercises 612

20 C++11 Regular-Expression Library 613
20.1 Overview of C++11 Regular Expressions 613
20.2 Dealing with Escape Sequences (\) 616
20.3 Constructing a RegEx String 618
  20.3.1 Matching Characters 618
  20.3.2 Pattern Modifiers 620

20.3.3 Recurring Groups 622
20.3.4 Character Classes 623
20.4 Matching and Searching Functions 624
20.5 “Find All,” or Iterative, Searches 626
20.6 Replacing Text 628
20.7 String Tokenizing 630
20.8 Catching RegEx Exceptions 631
20.9 Sample App: RPN Calculator 632
Exercises 635

A A Painless Introduction to Rvalue References (C++11) 637
A.1 The Trouble with Copying 637
A.2 Move Semantics: C++11 to the Rescue! 640
A.3 Rvalue Refs in a User’s String Class 642
A.4 Verifying Runtime-Performance Improvement 645
A.5 Rvalues and Contained Objects 646
A.6 References Reconsidered: Rvalues and Lvalues 646

B Summary of New Features in C++11 649
B.1 Improvements in Object Construction 649
B.2 Other Core-Language Enhancements 650
B.3 Other New Keywords 651
B.4 Extensions to the Standard Library 652

C ASCII Codes 655

Index 659

Preface

Congratulations! With this book, *C++ for the Impatient,* you have in your hands the quickest and easiest way to learn the latest state-of-the-art features in C++, especially features in the new C++11 specification and other features added in recent years.

The new technology is not difficult to understand once it's explained clearly. But some of the concepts are at first so abstract and alien that it can take weeks of pounding your head against a wall, wondering when you're going to "get it." Too many descriptions (although written by knowledgeable experts) are unable to break through the conceptual barrier that makes it hard to learn something new.

I wrote this book in part so that you wouldn't have to go through those weeks and months of frustration. Learning this material shouldn't be difficult.

C++ for the Impatient also provides quick, ready reference to all of C++'s features (or at least the great majority of them), ranging from the most basic building blocks of the language, up to and including refined uses of the C++ library.

How This Book Is Different: Learning C++11

My statement that this book is "the quickest and easiest way to learn the latest, state-of-the-art features" is a strong claim. It's backed up by a number of techniques.

- Dozens of easy-to-follow figures. Sometimes a picture really is worth a thousand words. Or sometimes ten thousand.
- Simple, relevant examples. Many or most programming books are written in part to convince you how smart the author is. My goal is to make *you* feel smart. And that means short, easy-to-digest examples.
- Frequent use of syntax diagrams, summarizing grammar at a glance.
- Simple explanations of important concepts—but only to the extent necessary to understand why a feature was added to C++ and how it's used.

Who Should Buy This Book?

The quick answer: almost anyone serious about learning and using C++.

- For advanced beginners: If you haven't learned any programming language before, you'll need additional help from a teacher or (at minimum) a more elementary text. But Chapter 1, "C++ Fundamentals," and the Sample Apps in each chapter are intended to be especially helpful for beginners.
- For intermediate and advanced programmers: No matter how advanced you may be, it's likely you're still coming up to speed on at least some of the newer and more advanced features of C++, especially C++11. If that's the case, this book is for you.
- For all programmers: Unlike other books, *C++ for the Impatient* features a concise write-up on each operator and each individual function, class, and object in the C++ library, including
  - Syntax summaries, showing arguments and types at a glance
  - A concise description of what each function, object, or operator does
  - A short illustrative example (although longer examples are used where helpful)

Examples and Exercises

Having the right example is often what makes the difference in learning new technology. So, in addition to the short examples for each individual function and operator, I provide one or two complete programs—Sample Apps—at the end of most chapters.

Each of these programs does something entertaining, useful, or both. They include intriguing games and puzzles. I've selected these Sample Apps because they make heavy use of the major technology in the chapter while at the same time being easy to follow.

Because this book is more a reference than a primer, I don't include exercises on every other page. However, I do include them after each Sample App. These are graduated and numbered, beginning with easier ones and leading up to the more truly challenging.

The best way to learn is by doing. I challenge you to complete the exercises in the book. If you can do them all, you can consider yourself quite an expert and an accomplished C++ programmer!

To find answers to these exercises (although it's cheating to look before you've at least tried to solve them yourself), you can go to informit.com/title/9780321888020. For information on other books of mine, you can go to brianoverland.com

Requirements: Your Version of C++

C++ is probably the most popular programming language in the world today. Although there are other excellent tools available for specific platforms (such as C#, Java, and Objective C), C++ still reigns as the most important, general-purpose language for writing and maintaining serious stand-alone applications.

Within the first few years after C++'s introduction, compiler vendors added important new features. This book assumes your compiler supports all of the following current features.

- Templates: This feature helps make code reusable; freeing you from having to solve the same general problems over and over again.
- The Standard Template Library (STL), providing flexible containers and reusable searching, sorting, and data processing routines: Potentially, these can save you a significant amount of work no matter what data types you work on.
- String class: The classic C-string type is still supported, but the STL **string** class is easier, safer, and more convenient to use than the old null-terminated C-string type.
- Specialized cast operators: These help result in more reliable, easier-to-debug programs.
- Structured exception handing: This is a superior way to handle errors, although it is mainly of use to programmers writing source code thousands of lines long.

One thing I do *not* assume in this book is that your compiler supports all the new features in the C++11 specification. As of this writing, some of the new features are not supported even by the major compiler vendors such as Microsoft, although they soon will be.

In order to make this book easier to use, therefore, all the sections and chapters that make use of the C++11 specification are clearly marked as such. The code in the rest of the book should run on any professional-grade compiler produced in the past several years.

C++11

Although this book doesn't assume that your compiler supports all the features in the C++11 specification, it is particularly useful if your compiler does support some or all of the new features.

The move from previous versions of C++ to C++11 is a major jump. Although the new specification is backward-compatible with past versions of C++, it opens up exciting new possibilities in software development.

This book explores all of these major new areas in C++11.

- Move semantics: The C++ library takes advantage of new technology by replacing copy operations, whenever possible, with *moves*. The difference is analogous to moving a document from one folder on your internal drive to another instead of copying it. You may have noticed that a move between folders is practically instantaneous, while copying can take time.

  Now this same improved performance is available to data within your own programs. Sometimes copying is unnecessary, and data transfers (or moves) are much faster. *Move semantics* can often speed up your programs without any action on your part. However, you can optimize your programs even further by facilitating move semantics in your own classes as explained in Appendix A, "A Painless Introduction to Rvalue References (C++11)."
- Lambda functions: Traditional programming techniques require you to define a function in one place and call it in another; this is fine most of the time, but sometimes it's inconvenient to search through all your source code to find a definition. Using the lambda technology enables you to define a function at the very point it is used, which can be extremely convenient.
- Improved syntax for class declarations: The ability to declare classes is one of the most important parts of the C++ language. Although C++ has been around for some time, up until now some things have been more difficult than they ideally should be. C++11 provides significantly improved syntax, such as inherited constructors for derived classes and more convenient ways to specify default values.
- Extended features of the C++ library: C++11 brings many new capabilities to the standard library, including a full-featured regular-expression library, which provides powerful tools for searching, matching, and replacing patterns of text. There are many other new capabilities as well, including improved randomization techniques and unordered (hash-table) containers. Such containers—although they lose the ability to step through elements in a meaningful order—provide much faster lookup times for database applications and data dictionaries. This is another way C++11 can significantly increase program performance.
- Smart pointers: Pointers are an important part of the C++ language, and they get their full due in this book, but applications can be made more error-free and easier to manage by the use of smart pointers, which automatically release memory when they go out of scope. This book describes exactly how the new **shared_ptr** and **unique_ptr** types work.
- Improvements to the core language: Finally, C++11 provides a series of improvements to the core language, such as range-based **for**, which reduces the chance of errors and is more concise than standard **for** statements. Range-based **for** takes advantage of each container's built-in knowledge of where it begins and ends. C++11 provides other improvements as well.

As Bjarne Stroustrup has said about C++11, it's effectively "a whole new language." Learning the new capabilities will enable you to produce higher-quality, more efficient and bug-free programs. You should find that the rewards, over time, repay the effort in learning these new features many times over.

Appendix B, "Summary of New Features in C++11," describes all these new features in greater detail.

Appendix C, "ASCII Codes," presents ASCII codes according to decimal and hexidecimal values.

Learning about Object Orientation

The approach of this book is to clearly lay out the mechanics of classes and objects in C++, explaining computer science concepts where needed to understand the technology.

For a more in-depth discussion of the background to object orientation and the history of C++, I recommend Bjarne Stroustrup's *The C++ Programming Language, Fourth Edition* (Addison-Wesley, 2013), which is long but is an excellent resource.

Other texts are available that attempt to teach how to "think objects" in more depth than I do in this book—although again, I cover the basic concepts. *C++ Primer, Fifth Edition* (Addison-Wesley, 2013) by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo, is a good place to start.

If you are really starting from scratch, with no background in C, C++, or a language in the C family, I humbly recommend my own book, *C++ Without Fear, Second Edition* (Prentice Hall, 2011).

Typographic Conventions

Typographic conventions for syntax summaries, I've found, can be helpful as long as those conventions don't get out of hand. In this book, I adhere to light use of these conventions. See Table P.1.

Throughout the book, I make extensive use of bold and italics in syntax displays; but I use brackets to indicate optional items only occasionally, because these can be confused with literal brackets. But the text always clarifies how I intend to use them.

Here is an example of these syntax conventions taken from Chapter 2, "Data":

[*sign*]*digits***.***digits***E***exponent*

In this case, both the decimal point (**.**) and the **E** are intended literally and so are in bold font. Each set of *digits* is a sequence of the numerals 0 through 9. The *exponent* is likewise a series of digits. Each of these is a string of numerals that you supply. The brackets (which are not in bold in this case) indicate that the *sign* is an optional minus sign (–) that can either appear or not appear.

TABLE P.1 Typographic Conventions

Style	Description
Keyword	Keywords and punctuation in bold are meant to be typed exactly as shown. Note that C++ is case-sensitive; so for the keyword **if**, enter "if" into your programming code but not "If" or "IF."
Placeholder	Placeholders are in italics: These are items that you supply. Typically, they are function, class, or variable names that you choose, but they can be something more complex, such as an entire statement.
[optional]	Occasionally, I use brackets (not in bold!) to indicate an optional item. In this case, the brackets are not intended literally but indicate something that may or may not be included as you choose. However, brackets in bold are intended literally, and in those cases, I mention that they are intended that way.
item**,** item…	Ellipses indicate an item that may be repeated any number of times. (Note that the comma is intended literally, but the ellipses are not.)

A Final Word

When Alice followed the White Rabbit down the rabbit hole, she was at first bewildered by the strange people and animals she encountered. So it is when learning new concepts. You wonder if you ever should have left your quiet garden and drawing room behind with their comfortable surroundings, as you encounter characters who say odd things and use peculiar logic.

You may find yourself asking: What was wrong with the old, familiar world I left behind? But hopefully, like Alice, you'll find yourself in Wonderland.

Personally, I feel the trip should always be interesting and even fun. Because this book is a reference, there are some chapters you won't want to read from beginning to end, any more than you'd read an encyclopedia that way. But the sections on the new technology are meant to be read like articles, as well as being texts you can come back to and refer to again and again. Above all, I hope they communicate a feel for why I found the new concepts both useful and intriguing.

Like me, you may be impatient. There is so much to learn and only so much time to learn it. I know that I'm like the White Rabbit, wanting to understand all that I see at first glance so that I can use the new tools to get to my destination quickly. I don't want to stand around trying to figure out what I'm doing there.

C++11 is almost a new language, despite its being backward-compatible. And learning a new language (to continue the Wonderland analogy) is like slaying the mythical Jabberwocky. It's a strange, multiheaded, whimsical beast, and it looks like it's too big to ever conquer; but with the right sword in hand, you shouldn't be afraid. Before you know it, you'll have completed the quest. And then you can imagine me saying:

"And, hast though slain the Jabberwock?
Come to my arms, my beamish boy!
Oh frabjous day! Callooh! Callay!"
He chortled in his joy."

This page intentionally left blank

Acknowledgments

More than any project I've worked on, this book required the help of some very talented people. To begin with, this book is the vision of editor Peter Gordon as much as it is mine, and it wouldn't have happened without his support and encouragement.

I was fortunate in getting assistance from some of the smartest people at Microsoft. Herb Sutter and Marian Luparu provided the software I needed as well as invaluable pointers. Stephan T. Lavavej provided insights on the STL (and yes, those are really his initials as well as his area of expertise!) that improved the book a great deal. John R. Bennett, a Microsoft emeritus, patent holder, and coauthor of Word Spellchecker, raised some questions on lambdas and regular expressions that enabled me to make the corresponding chapters much better. Ken Nichols, one of Microsoft's best technical minds, provided free reviews as well. All these people gave me advice and assistance without asking for any compensation whatsoever; an acknowledgment is the least I can do, even if that repayment is meager.

I'm indebted to Daveed Vandevoorde and Seve Vinosky for supplying superb technical reviews. They had an enormous amount of material to cover, and they did a great job.

Finally, I'm grateful to the excellent editorial team that showed patience, flexibility, and kindness to a sometimes-curmudgeonly author: Kim Boedigheimer, Caroline Senay, Audrey Doyle, and Alan Clements. Alan's cover art was so exceptional that it inspired me to make changes to the text itself to match the cover-art theme, something I've never done before.

This page intentionally left blank

About the Author

With long experience in both technology and training people to use it, **Brian Overland** is uniquely qualified to write books that simplify difficult concepts. He began programming professionally in C in the 1980s, working on a software-driven irrigation system used all over the globe. He also taught programming and English composition at a community college while writing film and drama reviews. At Microsoft, he spent a decade rising from tester and tech-support specialist to project lead and manager. As a project lead of Visual Basic 1.0, he played a key role in bringing easy Windows programming to the world and explaining how to use it; he was also a member of the Visual C++ team. He has since written many successful books and founded his own high-tech company.

This page intentionally left blank

◆ ◆ ◆ CHAPTER 20

C++11 Regular-Expression Library

Applications such as Microsoft Word have long supported pattern-matching, or *regular-expression,* capabilities. Using C and C++, it was always possible to write your own regular-expression engines, but it required sophisticated, complex programming—and usually a degree in computer science. Now C++11 makes these capabilities directly available to you, without your having to write a regular-expression engine yourself.

Regular expressions are of practical value in many programs, as they can aid with the task of *lexical analysis*—intelligently breaking up pieces of an input string—as well as tasks such as converting from one text-file format (such as HTML) to another.

I've found that when the C++11 regular expression library is explained in a straightforward, simple manner, it's easy to use. This chapter doesn't describe all the endless variations on the regex function-call syntax, but it does explain all the basic functionality: how to do just about anything you'd want to do.

20.1 Overview of C++11 Regular Expressions

Before using any regular-expression functions, include the <regex> header.

```
#include <regex>
```

A regular expression is a string that uses special characters—in combination with ordinary characters—to create a text pattern. That pattern can then be used to match another string, search it, or identify a substring as the target of a search-and-replace function.

For example, consider a simple pattern: a string consisting only of the digits 0 through 9, and nothing else. A decimal integer, assuming it has no plus or minus sign, fulfills this pattern.

With the C++11 regular-expression syntax, this pattern can be expressed as:

```
[0-9]+
```

In this regular-expression pattern, only the “0” and “9” are intended literally. The other characters—“[”, “]”, “-”, and “+”—each have a special meaning.

The brackets specify a range of characters:

`[`*`range`*`]`

This syntax says, “Match any one character in the specified range. The following examples specify different ranges.

```
[abc]          // Match a, b, or c.
[A-Z]          // Match any letter in range A to Z.
[a-zA-Z]       // Match any letter.
```

The other special character used in this example is the plus sign (+).

`expr+`

This syntax says, “Match the preceding expression, *expr*, one or more times. The plus sign is a pattern modifier, so it means that *expr*+, taken as a whole, matches one or more instances of *expr.*

Here are some examples:

```
a+            // Match a, aa, aaa, etc.
(ab)+         // Match ab, abab, ababab, etc.
ab+           // Match ab, abb, abbb, abbbb, etc.
```

Notice what a difference the parentheses make. Parentheses have a special role in forming *groups*. As with braces and the plus sign (+), parentheses are special characters; they have to be “escaped” to be rendered literally—that is, you have to use backslashes if you want to match actual parentheses in the target string.

You should now see why “[0-9]+” matches a string that consists of one or more digits. This pattern attempts to match a single digit and then says, “Match that one or more times.” Again, the plus sign is a pattern modifier, so it matches [0-9] one or more times *instead of* matching it just once, not one or more times *in addition to* matching it once (which would’ve meant a total of two or more times overall).

The following statements attempt to match this regular-expression string against a target string. In this context, *match* means that the target string must match the regular-expression string completely.

```
#include <regex>
#include <string>
#include <iostream>
. . .
```

```
std::regex  reg1("[0-9]+");
if (std::regex_match("123000", reg1)) {
     std::cout << "It's a match!" << std::endl;
}
```

You can test a series of strings this way:

```
using std::cout;
using std::endl;
using std::regex;

regex  reg1("[0-9]+");
string str1("123000");
string str2("123000.0");

bool b = regex_match(str1, reg1);
cout << str1 << (b ? " is " : " is not ");
cout << "a match." << endl;

b = regex_match(str2, reg1);
cout << str2 << (b ? " is " : " is not ");
cout << "a match." << endl;
```

These statements print out:

```
123000 is a match.
123000.0 is not a match.
```

The string “123000.0” does not result in a match because **regex_match** attempts to match the entire target string; if it cannot, it returns **false**. The **regex_search** function, in contrast, returns **true** if any substring matches. Therefore, the following function call returns **true**, because the substring consisting of the first six characters matches the pattern specified earlier for reg1.

```
std::regex_search("123000.0", reg1)
```

Generally speaking, every regular-expression operation begins by initializing a **regex** object with a pattern; this object can then be given as input to **regex_match** or **regex_search**. Creating a regex object builds a regular-expression engine, which is compiled at runtime (!), so for best performance, create as few new regular expression objects as you need to.

Here are some other useful patterns:

```
using std::regex;
regex reg2("[0-9]*");           // Match 0 or more digits.
regex reg3("(\\+|-)?[0-9]+");   // Match digit string with
                                //  optional + or - sign.
```

reg2 uses an asterisk (*) rather than a plus sign (+). The asterisk modifies the regular expression to mean, "Match zero or more copies of the preceding expression." Therefore, reg2 matches an empty string as well as a digit string.

reg3 matches a digit string with an optional sign. The "or" symbol (|) means match the expression on either side of this symbol:

```
"a|b"    // Match a or b but not both.
```

Putting "a|b" into a group (using parentheses) and then following it with a question mark (?), makes the entire group optional.

The following expression means, "Optionally match a plus sign or a minus sign, but not both."

```
"(\\+|-)?"
```

Because the plus sign (+) has special meaning, it must be "escaped" by using backslashes. More about that in the next section.

Note

There's a more concise way to write the expression in this case. Most symbols lose their special meaning inside brackets and do not need to be escaped. Even the minus sign (-) does not need to be escaped unless it occurs in between two other characters, indicating a run of characters (as in "[a-z]"). So the pattern just shown could be more concisely expressed as:

```
"[+-]?"
```

The limitation of a range, however, is that it is used just to match a single character, so the range syntax, [], is less general than "or" (|).

20.2 Dealing with Escape Sequences (\)

Escape sequences are a little tricky in C++ regular expressions, because they occur in two contexts.

- C++ assigns special meaning to the backslash within a string literal and requires it to be escaped to be read as an actual backslash: To represent a single backslash, it's necessary to place double backslashes (\\) in the source code. (Exception: Raw literals, supported by C++11, remove the need to escape characters.)
- The regular-expression interpreter also recognizes a backslash as the escape character. To render a special character literally, you must precede it with a backslash (\).

Consequently, if you want to render a special character literally, then, within a C++ literal string, you must precede the character with *two backslashes,* not just one.

For example, suppose you want to specify a pattern that matches an actual plus sign (+). The pattern is specified in source code this way:

```
std::regex  reg("\\+");
```

When the C++ compiler reads the literal, "\\+", it interprets \\ as an escape sequence that represents a single backslash. The actual string data that gets stored in memory is therefore:

```
\+
```

This is the string read by the regular-expression interpreter. It interprets "\+" as an actual plus sign (+).

Consider the following regular-expression pattern:

```
std::regex  reg("\\++");
```

Notice what's going on here: The first three characters (\\+) represent a literal plus sign (+). The fourth character (+) has its usual—and special—meaning; this second plus sign modifies the overall pattern to mean, "Match one or more copies of the preceding expression." The string as a whole therefore matches any of the following:

```
+
++
+++++
```

How do you represent a literal backslash, should you ever need to do that? That is, what is the regular-expression pattern that matches a target string consisting of one or more backslashes? The answer is that you need *four* backslashes.

```
using std::regex;
regex  reg("\\\\+");   // Matches one or more backslashes.
```

This regular-expression object, reg, would match any of the following:

```
str1[] = "\\"         // Represents "\".
str2[] = "\\\\"       // Represents "\\".
str3[] = "\\\\\\"     // Represents "\\\".
```

Note that if you use raw-string literals, supported by the C++11 specification, you don't have to deal with C literal-string escape conventions, so this example would be coded as:

```
regex  reg(R"\\+");    // Matches one or more backslashes.
str1[] = R"\"          // Represents "\".
str2[] = R"\\"         // Represents "\\".
str3[] = R"\\\"        // Represents "\\\".
```

The use of **R** does not change the format of the strings (which is still **const char***); it merely changes how literal text inside the quoted string is interpreted.

20.3 Constructing a RegEx String

The previous two sections provided an introduction to regular-expression patterns. The next several sections summarize the syntax rules, beginning with the syntax for matching individual characters.

This chapter adopts the default grammar used by the C++11 regular-expression library, which is a modified ECMAScript grammar. Although it's possible to use variations, the C++11 default is more versatile and expressive than the alternative grammars.

20.3.1 Matching Characters

The following special expressions match an individual character belonging to a group, such as letters or digits. This section also describes special conditions such as beginning-of-line or word boundary.

In the following list, a *range* may be a list of characters (not separated by spaces or commas, which themselves are characters). A *range* may optionally use a dash (minus sign) to indicate a run beginning with one character, up to and including another. Characters are ordered according to their underlying numeric (ASCII) value. For example, "[a-z]" matches all lowercase letters.

- `.`

  Matches any one character other than a newline. For example, the following pattern string matches almost any single character:

  `"."`

- **`[`*`range`*`]`**

  Matches any one character in the specified range. For example, the following pattern string matches any single letter in the range "a" to "m". It also matches "z".

  `"[a-mz]"`

  Most characters lose their special meaning inside the brackets. The minus sign gains special meaning to indicate a run of characters as in the example just shown, but only if it appears between two other characters inside the range. The following expression matches any one of the characters "+", "*", "/", or "-". None of these need to be escaped.

```
"[+*/-]"
```

- **[^*range*]**

  Matches any character *not* in the specified range. For example, the following pattern string matches any single character *other* than "a", "b", or "c":

  ```
  "[^abc]"
  ```

- **\n**

  Matches a newline. When using this in a C++ literal string meant to be part of a regular-expression pattern-matching string (as opposed to an actual embedded newline), remember that two backslashes must be used. For example:

  ```
  "\\n"
  ```

- **\t**

  Matches a tab character.

- **\f**

  Matches a form feed.

- **\r**

  Matches a carriage return.

- **\v**

  Matches a vertical tab.

- **\x*hh***

  Matches a character specified as a hexadecimal code. For example:

  ```
  "\\xf3"
  ```

- **\u*hhhh***

  Matches a Unicode character specified as a hexadecimal code. For example:

  ```
  "\\u02f3"
  ```

- **\d**

  Matches any digit character. This is equivalent to:

  ```
  [0-9]
  ```

- **\D**

  Matches any character other than a digit. This is equivalent to:

  ```
  [^0-9]
  ```

- **`\s`**

  Matches any whitespace character.

- **`\S`**

  Matches any character other than a whitespace character.

- **`\w`**

  Matches any digit, letter, or underscore.

- **`\W`**

  Matches any character other than a digit, letter, or underscore.

- **`\b`**

  Matches a word boundary. A word must begin or end at this position, or there is no match. Words are made up of alphanumeric characters and are delimited by whitespaces and punctuation. For example, the following string matches any word beginning with “c” and ending with “t”, such as “cat” or “containment”. It does not match “caution”.

  ```
  "\\bc[a-zA-Z]*t\\b"
  ```

- **`\B`**

  Not a word boundary. For example, the following pattern matches a portion of a word beginning with “a.” It will match “at” embedded in “cat”, but it will not match “at” if it occurs as a stand-alone word.

  ```
  "\\Ba[a-zA-Z]*"
  ```

- **`^`**

  Beginning of line: The next character is matched only if it is the first character in the text to be examined or occurs just after a newline.

  ```
  "^Barney"
  ```

- **`$`**

  End of line: The previous character matched (if any) must be the last character in the line of text.

20.3.2 Pattern Modifiers

Regular-expression pattern matching becomes more interesting when you modify a pattern to indicate possible repetitions. This feature, as much as anything else, makes the regular-expression technology a powerful and versatile tool for searching and replacing text.

In the following list, *expr* is an expression. For example, in the string “[0-9]+”, “[0-9]” is an expression and the + operator modifies its meaning.

An operator associates with the character closest to it, except where brackets or parentheses are used, in which case the operator refers to the whole range or group that precedes it.

- *`expr`*`*`

  Matches zero or more instances of *expr*. For example, the following string matches an empty string or a digit string:

  ```
  "[0-9]*"
  ```

- *`expr`*`+`

  Matches one or more instances of *expr*. For example, the following string matches a digit string of length one or greater.

  ```
  "[0-9]+"
  ```

- *`expr`*`?`

  Matches either one or zero instances of *expr*. The *expr* thereby becomes an optional item that can appear at most once. For example, the following string matches a minus sign or an empty string.

  ```
  "-?"
  ```

- *`expr1`*`|`*`expr2`*

  Matches *expr1* or *expr2*, but not both. For example, the following regular-expression string matches “aa” or “bb” but not “aabb”.

  ```
  "(aa)|(bb)"
  ```

  This expression can be made optional by placing it in a larger group and then using the ? operator. In that case, “aa” may appear, “bb” may appear, or they may both be omitted.

  ```
  "((aa)|(bb))?"
  ```

- *`expr`*`{`*`n`*`}`

  Matches exactly *n* instances of *expr*. For example, the following pattern string matches a target string containing exactly ten copies of capital “A”.

  ```
  "A{10}"
  ```

- *`expr`*`{`*`n`*`,}`

  Matches *n* or more instances of *expr*. For example, the following pattern string matches a target string consisting of three or more digits.

  ```
  "[0-9]{3,}"
  ```

- `expr{n,m}`

  Matches at least *n*, but no more than *m*, instances of *expr*. For example, the following pattern string matches a digit string no more than seven digits long.

  ```
  "[0-9]{1,7}"
  ```

- `(expr)`

  Forms a group. *expr* is considered as a unit when modified by other special characters, as in (*expr*)+, (*expr*)*, and so on. For example, the following pattern string matches "AbcAbcAbc" in the target string:

  ```
  "(Abc){3}"
  ```

  Another important effect of parentheses is that they cause the expression inside to be "tagged," as explained in the next section.

20.3.3 Recurring Groups

Much of the power of regular expressions comes from the ability to look for repetitions of a group. The syntax:

```
\n
```

refers to a previously tagged group. The expressions \1, \2, and \3 refer to the first three groups. Remember that C++ string literals use the backslash as an escape character, so the expressions "\1", "\2", and "\3" must be rendered as \\1, \\2, and \\3, and so on, in C++ source code (unless you're using raw string literals).

For example, the following expression—expressed as a string literal—matches aa, bb, and cc:

```
"(a|b|c)\\1"
```

This expression first matches a, b, or c. Whatever is matched is *tagged*. The regex pattern must then immediately match this tagged character again if it is to match the overall expression.

It will therefore match aa and bb, but not ab.

The next example is more practical: It finds a repeated word, in which a single space separates the two words:

```
"([A-Za-z]+) \\1"
```

This expression says, "Match a series of one or more letters. Tag the characters in this group. Then match a space. Finally, match an exact recurrence of the tagged characters." The following strings would therefore be matched:

```
"the the"
"Monday Monday"
"Rabbit Rabbit"
```

20.3.4 Character Classes

The C++11 grammar also provides a series of character classes that can be used to help specify a range. For example, the following expression specifies a range consisting of any letter:

```
[[:alpha:]]
```

This is equivalent to:

```
[A-Za-z]
```

The following expression specifies a range consisting of any letter or punctuation character:

```
[[:alpha:][:punct:]]
```

Descriptions of the character classes follow.

- **`[:alnum:]`**
  Any letter or digit.
- **`[:alpha:]`**
  Any letter.
- **`[:blank:]`**
  A space or tab character.
- **`[:cntrl:]`**
  Any control character. (These are not printable.)
- **`[:digit:]`**
  Any decimal digit.
- **`[:graph:]`**
  Any printable character that is not a whitespace.
- **`[:lower:]`**
  Any lowercase letter.
- **`[:print:]`**
  Any printable character, including whitespaces.

- **`[:punct:]`**

  Any punctuation character.

- **`[:space:]`**

  A whitespace character, such as a blank space, tab, or newline.

- **`[:upper:]`**

  Any uppercase letter.

- **`[:xdigit:]`**

  A hexadecimal digit: This includes digits, as well as uppercase and lowercase letters.

20.4 Matching and Searching Functions

This section summarizes the syntax for the matching and searching functions. Don't forget to include the <regex> header. Also, remember that regex symbols are part of the **std** namespace, so either include a **using** statement or identify each name with its **std** prefix.

```
#include <regex>
using std::regex;
```

Next, you need to specify a regular-expression pattern by creating a **regex** object:

```
regex  name(pattern_string [,flags])
```

This regex constructor includes an optional *flags* argument. The most useful flag is **regex_constants::icase**, which turns off case sensitivity.

regex objects are unusual in that they are compiled and built at runtime, not compile time. Consequently, you want to create as few **regex** objects as possible, because creating many such objects impacts runtime performance. You should make your **regex** objects global variables, or pass them by reference if you need to.

After constructing a **regex** object, you can perform matching and searching by calling the **regex_match** and **regex_search** functions. Each of these returns a Boolean value. The **regex_match** function returns **true** if the *target_string* matches the pattern stored in *regex_obj* exactly: That is, the entire *target_string* must match the pattern completely.

```
regex_match(target_string, regex_obj)
```

The **regex_search** function returns **true** if the target string contains any substring that matches the pattern stored in *regex_obj*.

regex_search(*target_string*, *regex_obj***)**

For example, consider the task of finding a repeated word in the sentence:

```
The the cow jumped over the the moon.
```

The following statements execute this search:

```
#include <regex>
#include <string>
#include <iostream>
using std::string;
using std::cout;
using std::endl;
. . .
std::regex reg1("([A-Za-z]+) \\1");
string target = "The the cow jumped over the the moon.";
if (std::regex_search(target, reg1)) {
     cout << "A repeated word was found." << endl;
}
```

The call to **regex_search** failed to find the first repeated-word occurrence ("The the") because the first word was capitalized and the second was not. That did not matter in this instance, because there was another repeated-word sequence ("the the"). However, sometimes you are interested in getting the position of the first match, so it can matter.

To find the position of the first substring found, it's necessary to include another argument:

regex_search(*target_string*, *match_info*, *regex_obj***)**

The *match_info* argument is an object of type **cmatch**, **smatch**, or **wmatch**, corresponding to the following formats: C-string, **string** object, and wide-character string. The format must match the type of *target_string*; in this case, the **string** class is used for the target string, so *match_info* must have type **smatch**.

For example:

```
std::smatch sm;
std::regex reg1("([A-Za-z]+) \\1");
std::string target="The the cow jumped over the the moon.";
bool b = std::regex_search(target, sm, reg1);
```

The *match_info* object (sm in this case) can be used to obtain information about the search, as follows:

```
match_obj.str()            // Return the matched string
match_obj.position()    // Return the position of the matched string
```

For example, the following call finds the position of the first repeated word:

```
std::smatch sm;
std::regex reg1("([A-Za-z]+) \\1");
std::string target="The the cow jumped over the the moon.";
bool b = std::regex_search(target, sm, reg1);
cout << "Match found at pos. " << sm.position() << endl;
cout << "Pattern found was: " << sm.str() << endl;
```

These statements print:

```
Match was found at pos. 24
Pattern found was: the the
```

Turning off case sensitivity causes a match to be found at position 0 instead.

```
std::regex reg1("([A-Za-z]+) \\1", regex_constants::icase);
```

You can also obtain information about groups—patterns enclosed in parentheses—within the matched string. This information only applies to groups within the *first* substring found; **regex_search** finds only one substring and then it stops searching. (The next section describes how to find multiple substrings.)

```
match_obj.str(n)        // Return nth group within the matched substring
match_obj.position(n)   // Return position of nth group
```

For example, you can use *match_object* (sm in this case) to get information on the group found *within* the first matched substring.

```
cout << "Text of sub-group: " << sm.str(1);
```

Assuming case sensitivity is turned off, this prints:

```
Text of sub-group: The
```

20.5 "Find All," or Iterative, Searches

When searching for substrings, you typically want to find all the matching substrings rather than just the first one. The easiest way to do this is to use an iterative search.

This technique involves creating two iterators: one that iterates through a target string and is associated with a **regex** object, and another that is simply an "end" condition.

```
sregex_iterator  iter_name(str_obj.begin(), str_obj.end(), regex_obj);
sregex_iterator  end_iter_name;
```

This second declaration—the one that creates *end_iter_name*—should strike you as something new: an uninitialized iterator that has a use. This is different from other STL containers, which require a call to their **end** function to get an end-position iterator. Not so with regex iterators: An uninitialized regex iterator automatically represents the ending position—one position past the end of the string.

The **sregex_iterator** type is an adapter that uses a template, **regex_iterator**, with the **string** class. You can build iterators upon other string types.

With these two declarations in place—providing an iterator and an end position—it's then an easy matter to iterate through all the substrings found. For example:

```
#include <regex>
#include <string>
using std::regex;
using std::sregex_iterator;
using std::string;

regex reg1("([A-Za-z]+) \\1", std::regex_constants::icase);
string target = "The the cow jumped over the the moon";
sregex_iterator it(target.begin(), target.end(), reg1);
sregex_iterator reg_end;
for (; it != reg_end; ++it) {
     std::cout << "Substring found: ";
     std::cout << it->str() << ", Position: ";
     std::cout << it->position() << std::endl;
}
```

These statements print:

```
Substring found: The the, Position: 0
Substring found: the the, Position: 24
```

This next example finds all words beginning with an uppercase or lowercase "B". Case sensitivity must be turned off in order to find "Barney" as well as "bat" and "big". In addition, the word-boundary character (\b) is used to ensure that the only patterns found are those in which the letter "B" comes at the beginning of the word.

```
regex reg2("\\bB[a-z]*", regex_constants::icase);
string bstr = "Barney goes up to bat with a big stick.";
sregex_iterator it(bstr.begin(), bstr.end(), reg2);
sregex_iterator reg_end;
for (; it != reg_end; ++it) {
     std::cout << "Substring found: ";
     std::cout << it->str() << ", Position: ";
     std::cout << it->position() << std::endl;
}
```

These statements print:

```
Substring found: Barney, Position: 0
Substring found: bat, Position: 18
Substring found: big, Position: 29
```

20.6 Replacing Text

One of the most powerful regular-expression capabilities is to selectively search-and-replace patterns within a string of text. Here's one possible use (out of zillions): to transform a target string by replacing each repeated pair of words with just one word.

For example, given this text:

```
The cow cow jumped over the the moon.
```

it would be useful to produce a string consisting of:

```
The cow jumped over the moon.
```

The **regex_replace** function performs this task by returning the transformed string. It has the following syntax:

`regex_replace(`*`target_string`*`, `*`regex_obj`*`, `*`replacement_pattern_str`***`);`**

The *replacement_pattern_str* is a string that can contain the following special sequences (in addition to ordinary characters).

- **`$&`**

  Refers to the entire matched string.

- **`$`***`n`*

  Refers to the nth group within the matched string. For example, "$1" refers to the first group of characters tagged by the regex object; "$2" refers to the second group of tagged characters (if there is one), and so on. The example that follows should clarify.

- **`$$`**

  A literal dollar sign ($).

The following declarations set up a search-and-replace designed to fix the repeated-word pattern, replacing it, where found, with one copy of the word.

```
using std::regex;
using std::regex_replace;
using std::string;
```

```
regex reg1("([A-Za-z]+) \\1");  // Find double word.
string replacement = "$1";      // Replace with one word.
```

With these objects defined, the following statements execute search-and-replace on the string shown earlier.

```
string target = "The cow cow jumped over the the moon.";
string result = regex_replace(target, reg1, replacement);
std::cout << result << std::endl;
```

The output is:

```
The cow jumped over the moon.
```

which is what we wanted.

Let's review how this works. When the text "cow cow" was matched by the regular-expression object, the first occurrence of "cow" was tagged because it matched the expression inside the parentheses: "([A-Za-z]+)". The rest of the expression, " \\1", indicated that the regex object then needed to match a space, followed by a recurrence of the tagged characters, to match the overall expression. Therefore, "cow" gets tagged and "cow cow" matches the entire regular expression.

The replacement pattern, "$1", causes the matched text—"cow cow"—to be replaced by "cow", the tagged group. Suppose the replacement pattern were "XX$1YY$1ZZ$1". Then the replacement text would have been "XXcowYYcowZZcow" and *that* would have replaced "cow cow".

Characters not matched by the regex object, reg1, are just copied into the result as they are. So, for example, the words "jumped over" are copied without being transformed.

Here's an example of another regex object and replacement-pattern string: When used with the call to **regex_replace** shown earlier, these result in the switching of two words separated by an ampersand (&). For example, "boy&girl" would be replaced by "girl&boy" and vice versa.

```
regex reg1("([A-Za-z]+)&([A-Za-z]+)");  // Find word&word
string replacement = "$2&$1";           // Switch order.
```

The **regex_replace** function is particularly convenient. It isn't necessary to iterate through the target string. Instead, **regex_replace** carries out replacements on all the substrings matching the pattern in the regex object, while leaving the rest of the text alone.

20.7 String Tokenizing

Although the functionality in the preceding sections can perform nearly any form of pattern matching, C++11 also provides string-tokenizing functionality that is a superior alternative to the C-library **strtok** function. Tokenization is the process of breaking a string into a series of individual words, or *tokens*.

To take advantage of this feature, use the following syntax, in which *str* represents a **string** object containing the target string:

```
sregex_token_iterator  iter_name(str.begin(), str.end(), regex_obj, -1);
sregex_token_iterator  end_iter_name;
```

As with **sregex_iterator**, **sregex_token_iterator** is an adapter built on top of the **string** class; you can use the underlying template, **regex_token_iterator**, with other kinds of strings.

sregex_token_iterator performs a range of operations, most of which are similar to what the standard iterator does, as described in Section 20.5, " "Find All," or Iterative Searches." Specifying -1 as the fourth argument makes the function skip over any patterns matching the *regex_obj*, causing the iterator to iterate through the tokens—which consist of text between each occurrence of the pattern.

For example, the following statements find each word, in which words are delimited by any series of spaces and/or commas.

```
#include <regex>
#include <string>
using std::regex;
using std::string;
using std::sregex_token_iterator;
. . .
// Delimiters are spaces (\s) and/or commas
regex re("[\\s,]+");
string s = "The White Rabbit,  is very,late.";
sregex_token_iterator it(s.begin(), s.end(), re, -1);
sregex_token_iterator reg_end;
for (; it != reg_end; ++it) {
     std::cout << it->str() << std::endl;
}
```

These statements, when executed, print the following, ignoring spaces and commas (except as to recognize them as delimiters):

```
The
White
```

```
Rabbit
is
very
late.
```

20.8 Catching RegEx Exceptions

Catching exceptions can be important when working with regular expressions, especially if you build a new regular-expression string at runtime or let the end user specify a pattern.

Remember that a regular expression object is compiled at runtime. If the pattern is ill-formed, the program throws an exception, causing abrupt termination unless the exception is caught. Catching the exception enables you to terminate more gracefully, or even—if you choose—continue execution after taking appropriate action (such as reporting the error to the user).

Regular-expression errors belong to the **regex_error** class, which is declared in the <regex> header and is part of the **std** namespace.

```
#include <regex>
using std::regex_error;
```

The **regex_error** class supports a **what** function, which returns a text string, and a **code** function, which returns an integer. The following example (which assumes the appropriate **#include** directives and **using** statements have been given) responds to a poorly formed **regex** object.

```
try {
     regex re("*\\bW[a-z]*");
     string s = "The White Rabbit is very late.";
     if (regex_search(s, re)) {
          cout << "Search string found." << endl;
     }
} catch (regex_error e) {
     cout << e.what() << endl;
     cout << "CODE IS: " << e.code() << endl;
}
```

When these statements are run, the program prints:

```
regex_error(error_badrepeat):
One of *?+{ was not preceded by a valid regular expression.
CODE IS: 10
```

The error is thrown as soon as there is an attempt to build a bad regular expression (because the regular-expression object, remember, is compiled at runtime).

```
regex re("*\\bW[a-z]*");  // Throws an exception!
```

Here you should be able to see the problem. The first use of the asterisk (*)—which means to repeat the preceding expression zero or more times—was not preceded by any expression; it was therefore not a meaningful use of regular-expression grammar. This is duly reported as a “badrepeat” error as long as exception handling is provided as just shown.

20.9 Sample App: RPN Calculator

In C++ *Without Fear,* 2nd Edition (Prentice Hall), I presented a Reverse Polish Notation (RPN) calculator as one of the more advanced examples. In this section, I present a superior version of that app.

An RPN calculator lets the user enter arbitrarily long arithmetic expressions in postfix notation. For example, to add 3 and 4, you specify not “3 + 4” but:

```
3 4 +
```

This might at first seem counterintuitive until you realize it’s an elegant notational system that does away with the need for parentheses. For example, the RPN expression:

```
3 4 + 10 1.5 + *
```

is equivalent to the following standard (infix) expression:

```
(3 + 4) * (10 + 1.5)
```

With RPN, an operator always applies to the expressions that precede it. In this case, the asterisk (*) applies to the expressions “3 4 +” and “10 1.5 +”, which produce 7 and 11.5, respectively. Multiplication is finally applied to produce 80.5. The RPN grammar can be summarized as:

expression <= *number*
expression <= *expression expression op*

Calculations: The stack mechanism, described in Section 16.3, “The stack Template,” is what powers this application. When the program reads a number, it pushes that number onto the stack. When the program reads an operator, it pops the top two values off the stack, performs a calculation, and pushes the result back onto the stack.

Lexical analysis: It's easy enough to interpret a line of input in which spaces separate operators as well as numbers. The more challenging problem is to recognize operators as both tokens *and* separators so that some of the spaces are optional. For example, it would be desirable to interpret:

```
3 44*5  1.2+/
```

as if it were written as:

```
3 44 * 5 1.2 + /
```

The **strtok** function is inadequate for this task. So is the **regex_token_iterator** function. The solution is to use a **regex_iterator** and search for sequences of characters that constitute either of the following:

- A number, consisting of consecutive digits with or without a fractional portion, such as "3", "44", or "100.507"
- Any of several operators: +, -, *, or /

With this approach, it's not necessary to have spaces on either side of an operator, although spaces are freely permitted. The following will work just fine:

```
3 4+ 1 2+*
```

The code for the application follows.

```
#include <iostream>
#include <string>
#include <cctype>
#include <regex>
#include <stack>

using std::cout;         // Alternatively, you can use
using std::cin;          //  using namespace std;
using std::endl;
using std::string;
using std::regex;
using std::sregex_iterator;
using std::stack;

void process_token(string s);
stack<double> st;

main() {
     string instr;
     string num_pattern("[0-9]+(\\.[0-9]*)?");
     string op_pattern("[+*/-]");
```

```
    regex re(num_pattern + "|" + op_pattern);
    while (true) {
        cout << "Enter expression (or ENTER to exit): ";
        getline(cin, instr);
        if (instr.length() == 0) { break; }
        sregex_iterator it(instr.begin(), instr.end(), re);
        sregex_iterator it_end;
        for (;it != it_end; ++it) {
             process_token(it->str());
        }
        if (!st.empty()) {
             cout << "The value is: " << st.top() << endl;
        }
    };
    return 0;
}

void process_token(string s) {
    // If s contains any char that is NOT an op,
    //  consider it a number by default.
    if (s.find_first_not_of("+*/-") != s.npos) {
         st.push(atof(s.c_str()));
    } else {
         double op2 = st.top(); st.pop();
         double op1 = st.top(); st.pop();
         switch(s[0]) {
         case '+': st.push(op1 + op2); break;
         case '-': st.push(op1 - op2); break;
         case '*': st.push(op1 * op2); break;
         case '/': st.push(op1 / op2); break;
         }
    }
}
```

This program illustrates a couple of important features of regular-expression grammar. First, as mentioned earlier, special characters such as “*” and “+” do not need to be escaped when they occur inside brackets (although brackets themselves would need to be escaped to be treated literally, of course). Also, the minus sign (-) does not need to be escaped, because it does not occur between two other characters.

```
    string op_pattern("[+*/-]");
```

Another interesting feature is that order is potentially significant. The **regex** object in this application searches for a digit string first and *then* for an operator. This order makes Exercise 2 possible.

Exercises

1. The following string pattern almost works but fails to separate numbers from operators in some cases. (Fortunately, the sample application contains the correct pattern.) Without looking at the application, determine what's wrong with the following regular expression:

   ```
   [0-9]+(.[0-9]*)?
   ```

2. Add support for a leading minus sign so that "-100", for example, is interpreted as a negative number. Ambiguities arise if the minus sign (-) is simply a unary operator in addition to being a binary operator (which it already is). But if you make the minus sign an optional part of a digit string, the user can add spaces for clarity as needed. Questions: Why does this exercise depend on the **regex** object looking for a digit string first and operators second? What would happen if it looked for operators first?
3. In the sample application, numbers of the form "33" and "33.03" are both accepted; "0.333" and "3." are also. However, strings with no integer portion and no leading zero—such as ".333"—are not accepted. Revise the expression for number_pattern so that it accepts digit strings such as ".333" in addition to the other formats.
4. Extend the floating-point format even further, to accept the following scientific-notation formats:

   digits`E`*digits*
   digits.*digits*`E`*digits*

5. Each time an operator is processed, the application assumes there are at least two values on the stack. If there are less than two, the user has entered too many operators. Revise the application to handle this situation by reporting the error and then continuing rather than incorrectly popping the stack (which causes program failure if you let it happen). Hint: Check the size of the stack.
6. Add other operators, such as a binary ^ operator, to perform exponentiation: "2 3 ^" should produce 2 to the 3rd power. Also add the tilde (~) as a unary operator to perform arithmetic negation (multiplying by -1). Finally, add the == and != operators to perform test-for-equality and test-for-inequality. These operators, which are Boolean, should each return either 1 or 0.
7. As a *really* advanced exercise (not for the faint of heart), enable the RPN calculator to read symbols and assign values by using a single equal sign (=) as an assignment operator. Build a symbol table by using the **map** template. Whenever a symbol—that is, a name—appears in any context other than the left side of an assignment, look up its value in the map.

This page intentionally left blank

Index

Symbols

&. *See* Ampersands (&)
<> (angle brackets) for unique_ptr, 184–185
' (apostrophes)
    single-quoted characters, 39–40, 295
    special character, 43
* (asterisks). *See* Asterisks (*)
\ (backslashes). *See* Backslashes (\)
^ (carets)
    bitwise exclusive OR operator, 72, 76
    regular expressions, 619–620
: (colons). *See* Colons (:)
, (commas). *See* Commas (,)
{} (curly braces). *See* Curly braces ({})
$ (dollar signs) in regular expressions, 620, 628
. (dot) notation
    classes, 192
    member functions, 195, 198
    unions, 244
" (double quotation marks)
    special character, 43
    stringizing operator, 259
    strings, 6, 40–41, 293
= (equal signs). *See* Equal signs (=)
! (exclamation points)
    comparisons, 12
    negation, 64
> (greater than signs)
    comparisons, 12, 70–71
    shift operators, 8, 70, 76, 81
    stream-input operator, 4
< (less than signs)
    comparisons, 12, 70–71
    maps, 509
    shift operators, 8, 69–70, 76, 81
- (minus signs). *See* Minus signs (-)
(number signs)
    format specifiers, 316
    preprocessor directives, 250
() (parentheses)
    function calls, 121, 224–225
    operator precedence, 58
    regular expressions, 614
% (percent signs)
    modulus assignment operator, 68, 76
    printf format specifiers, 313–315
    scanf format specifiers, 317–319
+ (plus signs). *See* Plus signs (+)
? (question marks)
    conditional operator, 73
    pattern modifiers, 621
    regular expressions, 616
; (semicolons). *See* Semicolons (;)
' (single quotation marks)
    single-quoted characters, 39–40, 295
    special character, 43
/ (slashes)
    assignment operator, 76
    comments, 8, 17–18
    division, 10, 68
[] (square brackets). *See* Square brackets ([])
~ (tildes) for bitwise negation, 64
_ (underscores) in symbolic names, 30
| (vertical bars). *See* Vertical bars (|)

A

%a format specifiers, 342, 345
%A format specifiers, 345
a mode specifiers, 323
a+ mode specifiers, 323
\a special character, 42

abort function, 356
abs function, 336, 356
Absolute value functions, 336, 356–357
Access specifiers, 231–232
accumulate algorithm
    default behavior, 562
    detailed description, 565
acos function, 335
Adding machine app, 19–20
Addition
    assignment operator, 74–75
    example, 10
    overview, 69
    precedence, 58
Address operator (&)
    description, 65
    pointers, 159, 162–163
Addresses
    pointers. *See* Pointers
    strings, 295, 414–415
adjacent_difference algorithm
    default behavior, 562
    detailed description, 565–566
adjacent_find algorithm
    detailed description, 566
    purpose, 557
Aggregates
    data declarations, 47
    deques, 450
    initialization lists, 153
    list containers, 467
    nested, 158
    vector list initialization, 439
<algorithm> header, 137, 547, 556–558, 564
Algorithms, 547
    containers, 432
    descriptions. *See specific algorithms by name*
    general concepts, 547–550
    heap, 560–561
    insert iterators, 553–554
    iterator overview, 551–552
    lambda functions, 550–551
    median app, 555–556
    modifying, 556–558
    numeric, 562
    read-only, 556–557
    sorting and reordering, 558–559
Aliases
    data types, 52–53
    references. *See* References
all_of algorithm
    detailed description, 566–567
    purpose, 557
[:alnum:] character class, 623
[:alpha:] character class, 623
Alpha file organizer app, 461–463
Ampersands (&)
    address operator, 65, 159, 162–163
    bitwise AND operator, 72, 76
    copy constructors, 210
    lambda functions, 134–136
    logical AND operator, 73
    pointers, 159, 163, 175
    in printing, 284
    reference arguments, 101–102, 147–150
    regular expressions, 629
    rvalue references, 643
AND operator
    bitwise, 72, 76, 78–79
    logical, 73
Angle brackets (<>) for unique_ptr, 184–185
Anonymous functions. *See* Lambda functions
Anonymous unions, 244–245
any function, 460
any_of algorithm
    detailed description, 567
    purpose, 557
Apostrophes (')
    single-quoted characters, 39–40, 295
    special character, 43
app mode flag, 387
append function
    description, 411
    purpose, 410
    string iterator, 429
Append mode, 323, 387–388
Arguments
    calling functions, 121–122
    default values, 128–129
    defining functions, 120–121
    lambda functions, 133–134
    overloading functions, 126–128
    vs. pointers, 167
    pointers as, 162–163

  - prototyping functions, 119–120
  - references, 121, 147–150
  - variable-length lists, 129–131
- Arithmetic
  - basic operators, 10
  - iterators, 427–428, 436
  - negation operator, 65
  - pointers, 166–167
- Array arguments vs. pointers, 167
- Arrays, 152–153
  - char, 291
  - heap algorithms, 560–561
  - loops for, 154–155
  - multidimensional, 157–159
  - one-dimensional, 153–154
  - passing to functions, 155–156
  - pointer access to, 163–166
  - sieve of Eratosthenes app, 186–188
- asin function, 335
- assert predefined macro, 261
- assign function
  - deques, 452
  - lists, 472
  - string iterators, 429
  - strings, 411–412
  - vectors, 440–441
- Assigning strings, 406
- Assignment operator (=)
  - comparisons, 12
  - overview, 73–76, 222–224
  - rvalue references, 644
- Associated containers, 432–433
  - Guess-the-Word app, 543–546
  - map templates. *See* Maps and map templates
  - multimaps, 529–532
  - multisets, 532–534
  - pair templates, 495–497
  - sets. *See* set templates
  - unordered. *See* Unordered containers
- Associativity
  - operators, 58
  - pointers, 163–164
- Asterisks (*)
  - assignment operator, 75
  - comments, 17–18
  - multiplication, 10, 68
  - operator precedence, 58
  - pattern modifiers, 621
  - pointers, 65, 67–68, 159–165, 175–178
  - regular expressions, 616
  - scanf, 320
- at function
  - deques, 453
  - strings, 412
  - vectors, 441
- At operator (*) for pointers, 65, 159–164
- atan function, 335
- atan2 function, 335
- ate mode flag, 388
- atexit function, 356
- atof function, 317, 347, 365–366
- atoi function, 317, 347
- atol function, 347
- auto keyword
  - lambda functions, 133
  - uses, 47
  - working with, 53–54
- auto modifier, 47
- Automatic deletion of pointers, 181

B

- \b character in regular expressions, 620
- \B character in regular expressions, 620
- %b format specifiers, 345
- %B format specifiers, 345
- b mode specifiers, 323
- \b special character, 42
- back function
  - deques, 453
  - lists, 472
  - queues, 486
  - vectors, 441
- back_inserter function, 553–554
- Backslashes (\)
  - line-continuation characters, 249
  - pathnames, 322
  - regular expressions, 616–619
  - special characters, 41–45
- Backspace special character, 42
- bad_alloc exception
  - description, 108
  - with new, 65–66
  - pointers, 169
- bad_cast exception, 108
- bad function, 382

bad_typeid exception, 108
Base 8 format, 32
Base 16 format, 32
Base-class
    access specifiers, 231–232
    pointers, 233–235
beg flag, 394
begin function
    deques, 449–450, 453
    lists, 467–468, 473
    maps, 508, 512–513
    sets, 524
    strings, 425–426
    unordered containers, 539
    vectors, 436, 441
Beginning of line in regular expressions, 620
Bell special character, 42
bernoulli_distribution distribution, 606
Biased distributions, 599–600
Bidirectional iterators, 552
Big-endian systems, 242
Binary files
    reading and writing, 327–328
    vs. text, 385–386
Binary mode
    file streams, 391–392
    opening files in, 323
binary mode flag, 388
Binary operators
    arithmetic, 9–10
    member functions, 220
Binary printout app, 88–90
binary_search algorithm
    detailed description, 567
    purpose, 557
Binary search function, 352–353
Binary trees
    heap algorithms, 560–561
    maps, 507–509
binomial_distribution distribution, 606
bit_and function-object template, 564
Bit fields, 240–242
bit_or function-object template, 564
bit_xor function-object template, 564
Bitmask flags
    output streams, 383
    stream manipulator, 374–375
Bitset templates
    constructors, 459
    description, 433
    functions, 460–461
    overview, 458–459
Bitwise operators
    AND, 72, 76, 78–79
    exclusive OR, 72, 76, 78–79
    inclusive OR, 76, 78–79
    negation, 64
    shift, 76, 81
    signed and unsigned types, 79–81
[:blank:] character class, 623
Blocks, 91, 94–95
bool and Boolean data type
    description, 28
    format flags, 378
    vs. integer, 34–35
    literal values, 32
    packed Boolean app, 245–248
boolalpha manipulator, 371
Braces. *See* Curly braces ({})
Brackets
    angle (<>), 184–185
    square ([]). *See* Square brackets ([])
break statements
    description, 93
    overview, 104–106
    switch statements, 104
bsearch function, 173–174, 352–353
bucket function, 540
bucket_count function, 540
Buckets for unordered containers, 537–538
Buffer flushing, 330, 372–373, 376
Byte searches in memory, 305

C

%c format specifiers
    date and time, 345
    printf, 313
    scanf, 318
C-language cast operators, 86–87
C-language-style comments, 17–18
c_str function, 310, 387, 395, 414–415
C-strings
    double-quoted strings, 40–41

- functions, 294–299
- input and output, 293–294
- overview, 291–293

callback functions
- function-call operators, 224
- search, 174, 177, 351–352
- sort, 354
- virtual functions, 237

Calling functions
- member functions, 198–200
- overview, 121–122

calloc function, 349–350

capacity function
- strings, 412
- vectors, 441–442

Capture placeholders in lambda functions, 134

Carets (^)
- bitwise exclusive OR operator, 72, 76
- regular expressions, 619–620

Carriage returns
- regular expressions, 619
- special character, 43

Case-sensitivity
- searches, 416
- string comparisons, 408, 413
- symbol names, 30

case statements
- description, 92
- switch statements, 103–104

Cast operators, 82–83
- C-language_cast, 87–88
- const_cast, 85–86
- dynamic_cast, 86–87
- reinterpret_cast, 84–85
- static_cast, 83–84

catch statements
- exception objects, 109–110
- regular expressions, 631–632
- syntax, 108–109

cauchy_distribution distribution, 608

<cctype> header, 301, 412

ceil function, 336

cerr object, 368

char data type and characters, 27
- arrays, 291
- classes for regular expressions, 623–624
- format specifiers, 313, 318
- functions, 301–304
- literals. *See* String and character literals
- regular expressions, 618–620
- retrieving, 310, 325–326
- single-quoted, 39–40
- special, 41–45
- strings. *See* Strings
- wide, 33, 45–46, 306–307

chi_squared_distribution distribution, 608

Child objects, 233–235

cin object, 8–9
- cin.getline function, 294
- cin.ignore function, 4–5, 9
- istream class, 366–368
- with right-shift operator, 363–364
- strings, 294

class keyword
- declaring classes, 190–191
- description, 190
- with enum, 52

Class templates, 272–273
- instantiating, 274–275
- with member functions, 276–278
- simple, 273–274

Classes
- bit fields, 240–242
- calling member functions, 198–200
- constructors. *See* Constructors
- containing, 201–203
- data-member access, 193–194
- declaring and using, 191–193
- defining member functions, 195–198
- deriving. *See* Deriving classes
- destructors, 216–217
- I/O stream. *See* I/O stream classes
- incoming and outgoing conversion functions, 228–229
- member access, 63
- overview, 189–190
- packed Boolean app, 245–248
- private member functions, 200–201
- static members, 203–205

clear function
- deques, 453
- lists, 473
- maps, 513
- sets, 525

clear function (*continued*)
    streams, 382
    strings, 413
    vectors, 442
clearerr function, 330
Clearing error status, 330
clock function, 340–341
CLOCKS_PER_SEC constant, 340
clog object, 368
close function, 390
Closing
    files, 324
    streams, 390
Closure syntax for lambda functions, 133–136
cmatch type, 625
<cmath> header, 2, 100, 198, 333–334, 336, 356
[:cntrl:] character class, 623
Colons (:)
    conditional operator, 73
    constructors, 215
    member functions, 195–196
    public members, 193
    scope operator, 62
Comma operator, 58, 76–77
Command string execution, 358
Commas (,)
    aggregates, 47
    argument lists, 119, 139
    enumerated types, 50
    initialization lists, 153, 212
    variable declarations, 26
Comments, 8
    C-language-style, 17–18
    #define with, 254
    line, 17
Common randomization engines, 604–605
compare functions, 413
Comparisons and comparison functions
    C-strings, 355
    memory, 305
    operators, 12, 70–71
    strings, 296–297, 408, 413
    test for equality, 71–72
    vector iterators, 436
Compilation, conditional, 257–259
Compound statements, 91, 94–95
Computer guesses the number app, 113–116
Concatenating strings, 69, 293, 295, 297, 407
Conditional compilation, 257–259
Conditional operator, 73
Conditions
    control structures, 91
    if statements, 96
    if-else statements, 11–13
    while statements, 98
Console I/O functions, 310–313
Console object, 3
const_cast operator, 85–86
const_iterator type
    deques, 450
    lists, 469
    vectors, 438
const keyword, 47
    function declarations, 125
    initialization lists, 212–213
    pointers, 170–173
    references, 151–152
const_reverse_iterator type
    deques, 450
    lists, 469
    vectors, 438
constexpr keyword, 48, 125, 141–142
Constructors, 205–206
    bitsets, 459
    copy, 210–211
    default, 206–208
    default member initialization, 215–216
    delegated, 214–215
    deques, 450–451
    inherited, 232–233
    initialization lists, 211–214
    lists, 469–470
    maps, 509–511
    move, 644
    overloaded, 208–210
    priority_queues, 489–490
    sets, 521–523
    string class, 405–406
    vectors, 438–439
Containers
    associated. *See* Associated containers
    description, 431
    iterators, 433–434
    range-based for statements, 101–102
    rvalue references, 646
    templates, 431–433

continue statement
  description, 93
  for statements, 99
  jump statements, 105
Control characters, testing for, 302
Control structures, 10
  compound statements, 94–95
  computer guesses the number app, 113–116
  exception handling, 106–111
  for statements, 99–101
  Guess-the-Number app, 111–113
  if and if-else statements, 11–13, 96–98
  jump statements, 104–106
  null statements and expression statements, 93–94
  range-based for statements, 101–102
  statement summary, 91–93
  switch statements, 103–104
  while and do-while statements, 13–14, 98–99
Conversions
  constructors, 208–210
  degrees to radians, 334
  incoming and outgoing, 228–229
  string-to-number functions, 347–348
copy algorithm
  detailed description, 567–568
  purpose, 558
copy_backward algorithm
  detailed description, 568
  purpose, 558
Copy constructors, 210–211
copy function, 414
copy_if algorithm
  detailed description, 568
  purpose, 558
copy_n algorithm
  detailed description, 568–569
  purpose, 558
copyfmt function, 382
Copying
  characters, 414
  deep, 210, 233
  memory, 304–305
  rvalue references in, 637–640
  strings, 296–297
Core-language new features, 650–651
cos function, 335
cosh function, 335
Cosine functions, 335
count algorithm
  detailed description, 569
  purpose, 557
count function
  bitsets, 460
  maps, 513
  multimaps, 529
  multisets, 533
  sets, 525
count_if algorithm
  detailed description, 569
  purpose, 557
cout object, 2, 8
  description, 3
  with left-shift operator, 362
  ostream class, 366–368
__cplusplus predefined macro, 262
<cstdio> header, 310, 321–322, 325
<cstdlib> header, 111, 347, 349, 351, 355
<cstring> header, 294, 304, 403–404
ctime function, 339, 341
<ctime> header, 11, 339–340, 342, 351, 601
<ctype> header, 301
cur flag, 394
Curly braces ({})
  aggregates, 439
  class declarations, 191
  compound statements, 91, 94–95
  enumerated types, 50
  function definitions, 120
  initialization lists, 153
  main function, 3
  member functions, 195
  pattern modifiers, 621–622
  semicolons after, 6, 9
  sets, 519–520
  while loops, 13
<cwchar> header, 306

D

\d character in regular expressions, 619
\D character in regular expressions, 619
%d format specifiers
  date and time, 345
  printf, 313
  scanf, 318
Dangling references, 140

Data, 25
  declarations, 46–50
  enumerated types, 50–52
  mixing numeric types, 33–38
  numeric literals, 31–33
  primitive types, 27–30
  simple variables, 25–26
  special declarations, 52–54
  string and character literals. *See* String and character literals
  symbolic names, 30
  type ambiguities, 269–270
  type promotion apps, 54–55, 288–289
Data-access operators, 62–64
Data dictionaries, 495
data function for strings, 415
Data members in classes, 193–194, 200
__DATE__ predefined macro, 261–262
Date and time functions, 339
  format specifiers, 344–346
  overview, 340–342
  random number seeds, 603
  tm structure, 342–344
Daylight saving time (DST) function, 344
dec manipulator, 370–371
Decimal format
  format specifiers, 313, 318
  stream manipulator, 371
Declarations
  classes, 191–193
  data, 46–50
  functions, 117–119, 124–126
  with pointers, 175–178
  project header files, 263–264
  special, 52–54
  strings, 292
  variables, 7–10, 25–26
decltype keyword, 54
Decrement operators
  classes, 227–228
  postfix, 63
  prefix, 64
Deep copying, 210, 223
default_random_engine object, 602, 604
default statements
  description, 92
  switch statements, 103
Defaults
  argument values, 128–129
  constructors, 206–208
  member initialization, 215–216
#define preprocessor directive
  description, 250–251
  enumerated types, 50
  macros, 256–257
  meaningful symbols, 254–255
defined operator, 260
Definitions
  functions, 117–121
  project header files, 263
  vs. prototypes, 195–198
Degrees measures, 334
Delegated constructors, 214–215
delete keyword
  objects, 66, 216
  pointers, 168–169
  support for, 348
  unique_ptr, 186
Deleting files, 330
deque templates
  constructors, 450–451
  description, 433
  functions, 451–458
  heap algorithms, 560
  iterators, 449–450
  overview, 447–449
  for queues, 484
  for stacks, 481
Dereference operator, 63, 65
Deriving classes, 229
  base-class access specifiers, 231–232
  inherited constructors, 232–233
  name conflicts, 239–240
  override keyword, 238–239
  pure virtual functions, 238
  subclass syntax, 229–231
  up-casting, 233–235
  virtual functions and overriding, 235–237
Destructors, 216–217
Dice apps
  odds, 142–145
  random functions, 610–612
Dictionaries, maps, 495
difftime function, 341

[:digit:] character class, 623
Digits
  regular expressions, 614, 619
  testing for, 302
Directives. *See* Preprocessor directives
discard function, 609
discrete_distribution distribution, 608
Distributions
  biased, 599–600
  operations on, 609–610
  randomization, 605–608
div function, 357
div_t structure, 357
divides function-object template, 563
Division
  assignment operator, 76
  example, 10
  overview, 68
do while statements
  description, 92
  overview, 98–99
Dollar signs ($) in regular expressions, 620, 628
domain_error exception, 108
DOS box, 4
Dot (.) notation
  classes, 192
  member functions, 195, 198
  unions, 244
double data type
  declaring, 7–8, 26
  description, 28
  format specifiers, 317, 320
  literals, 32
Double-ended queues. *See* deque templates
Double quotation marks (")
  special character, 43
  stringizing operator, 259
  strings, 6, 40–41, 293
DST (daylight saving time) function, 344
dynamic_cast operator, 86–87

E

%e format specifiers
  printf, 313
  scanf, 318
%E format specifiers
  printf, 314
  scanf, 318
Elapsed time function, 341
#elif preprocessor directive, 251
Ellipses (. . .) in variable-length argument lists, 129
empty function
  deques, 453
  lists, 473
  maps, 513
  priority_queues, 490
  queues, 486
  sets, 525
  stacks, 483–484
  strings, 415
  vectors, 442
end flag, 394
end function
  deques, 449–450, 453
  iterators, 434
  lists, 467–468, 473
  maps, 501, 513
  sets, 520–521, 525
  strings, 425–426
  unordered containers, 540
  vectors, 436, 442
End of line in regular expressions, 620
Endian-ness, 242
#endif preprocessor directive
  conditional compilation, 18, 257–259
  description, 251
endl manipulator, 6, 371, 396
ends manipulator, 371
Engines for randomization, 608–609
enum keyword, 50–52
Enumerated types
  bit fields, 240–241
  overview, 50–52
Environment variables, 357
eos manipulator, 396
equal algorithm
  detailed description, 569–570
  purpose, 557
equal_range algorithm
  detailed description, 570
  multimaps, 532
  purpose, 557

Equal signs (=)
  assignment operator, 222–224
  comparisons, 12
  if statements, 97
  lambda functions, 135–136
  rvalue references, 644
  strings, 406
  test for equality operator, 71–72
equal_to function-object template, 563
Equals functions, writing, 541–543
erase function
  deques, 453–454
  lists, 469, 473–474
  maps, 513–514
  sets, 525–526
  string iterators, 429
  strings, 415
  vectors, 435, 442–443
Eratosthenes sieve, 186–188
error output object, 368
#error preprocessor directive, 251
Errors
  clearing status, 330
  strings for, 297
Escape sequences (\)
  regular expressions, 614, 616–618
  special characters, 41–45
exception class, 107, 109–110
Exceptions and exception handling, 106
  catch blocks and exception objects, 109–110
  exceptions, 107–108
  regular expressions, 631–632
  throw operator, 76, 110–111
  try-catch blocks, 108–109
Exclamation points (!)
  comparisons, 12
  negation, 64
Exclusive OR operators
  assignment, 76
  description, 72
  detail, 78–79
exit function, 357
exp function, 336
explicit keyword
  constructors, 210
  conversion functions, 228
  function declarations, 125
exponential_distribution distribution, 606–607
Exponents
  format specifiers, 313–314
  functions, 337–338
  literals, 32
Expressions, 57
  description, 91
  overview, 93–94
  regular. *See* Regular expressions
extern modifier, 48
extreme_value_distribution distribution, 608

F

\f characters
  regular expressions, 619
  special character, 42
%f format specifiers
  printf, 314
  scanf, 318
fabs function, 198, 337
fail function, 382–383
false value, 33
fclose function, 324
ferror function, 330
fflush function, 330
fgetc function, 325
fgetpos function, 328
fgets function, 311, 325–326
Fibonacci numbers, 20–23
__FILE__ predefined macro, 262
File I/O, 321–322
  buffer flushing, 330, 372–373, 376
  closing, 324
  deleting, 330
  miscellaneous functions, 330–331
  opening, 322–324
  random-access functions, 328–329
  renaming, 330
  text, 325–327
File stream operations, 385
  binary mode, 391–392
  member functions, 390–391
  objects, 387–390
  random-access operations, 392–394
  text vs. binary files, 385–386
FILE type, 322

fill algorithm
  detailed description, 570
  purpose, 558
Fill character, 375
fill function, 381
fill_n algorithm
  detailed description, 570
  purpose, 558
final modifier, 125
find algorithm
  detailed description, 571
  purpose, 557
"Find All" searches, 626–628
find_end algorithm
  detailed description, 571
  purpose, 557
find_first_not_of function, 416–417
find_first_of algorithm
  detailed description, 571–572
  purpose, 557
  strings, 417
find function
  map elements, 501–504, 514
  set elements, 520–521, 526
  strings, 415–416
find_if algorithm
  detailed description, 572
  purpose, 557
find_if_not algorithm
  detailed description, 572
  purpose, 557
find_last_not_of function, 417–418
find_last_of function, 418–419
Finding
  map elements, 501–502
  set elements, 520–521
  substrings, 295–296, 415–416
First-in-first-out (FIFO) storage, 484–487
fisher_f_distribution distribution, 608
fixed manipulator, 371–372
Fixed-point format, 371–372
Flag-setting stream functions, 382–385
flags, format, 375, 377–378
flags function, 383
Flashing console, 4–5
flip function, 460
float and floating-point data, 28
  declaring, 26
  double, 8, 28
  format flags, 378
  format specifiers, 313–314, 318
  guidelines, 29–30
  vs. integer, 34
  literals, 32
  stream manipulator, 375–376
floor function, 337
flush function, 372, 381
Flushing file buffers, 330, 372–373, 376
fmod function, 68, 337
fopen function, 322–324
for_each algorithm
  detailed description, 573
  overview, 548–549
  purpose, 557
for statements
  arrays, 154–156
  break statements, 104–106
  description, 92
  overview, 99–101
Form feeds
  regular expressions, 619
  special character, 42
Format flags, 375, 377–378
Format specifiers
  date and time functions, 344–346
  printf, 313–315
  scanf, 317–320
Forward iterators, 552
fpos_t type, 328–329
fprintf function, 309, 326
fputc function, 326
fputs function, 325–326
Fraction function, 338
fread function, 328
Free exponent function, 337
free function, 349
freopen function, 330
frexp function, 81, 337
Friends
  data-member access, 194
  operator functions as, 221–222
front function
  deques, 454
  lists, 474
  queues, 487
  vectors, 443

front_inserter function, 553–554
fscanf function, 326
fseek function, 329
fsetpos function, 328–329
fstream class, 390
<fstream> header, 387
ftell function, 329
Function-call operator (), 224–225
Function templates, 267
    one parameter, 267–269
    two parameters, 270–272
    type ambiguities, 269–270
<functional> header, 139, 562
Functions, 117
    arguments with default values, 128–129
    bitsets, 460–461
    calling, 121–122, 198–200
    class members, 200
    comparison, 355
    constexpr, 141–142
    date and time, 339–346
    declarations, 117–119, 124–126
    defining, 120–121
    deques, 451–458
    dice odds app, 142–145
    lambda. *See* Lambda functions
    lists, 471–481
    local and global variables, 122–124
    maps, 511–518
    math, 336–339
    memory-allocation, 348–350
    object, 562–564
    operator, 63
    overloading, 126–128
    passing arrays to, 155–156
    pointers, passing and returning, 178–180
    predefined objects, 562–564
    priority_queues, 490–491
    prototypes vs. definitions, 117–119
    prototyping, 119–120
    randomization, 350–351
    returning references from, 150–151
    searching and sorting, 351–355
    set templates, 523–529
    string-to-number conversions, 347–348
    strings, 294–299
    trigonometric, 333–336
    variable-length argument lists, 129–131
    vectors, 439–447
Fundamentals, 1
    adding machine app, 19–20
    boilerplate program, 6–7
    comments, 17–18
    control structures, 10–14
    elements, 1–4
    flashing console, 4–5
    general program structure, 14
    Microsoft Visual Studio, 5
    namespaces, 15–17
    phi calculation app, 20–23
    variable declarations, 7–10
fwrite function, 328

G

%g format specifiers
    printf, 314
    scanf, 318
%G format specifiers
    printf, 314
    scanf, 318
gamma_distribution distribution, 608
Garbage collection, 180–181
Garbage variables, 8
gcount function, 379, 391
generate algorithm
    detailed description, 573
    purpose, 558
generate_n algorithm
    detailed description, 573
    purpose, 558
Generic pointers, 173
geometric_distribution distribution, 607
get function
    input streams, 379–380
    shared_ptr, 184
    tuples, 287
    unique_ptr, 185
getc function, 327
getchar function, 310
getenv function, 357
getline function
    C-strings, 294
    input streams, 380

- reading lines with, 364–366
- text files, 389

gets function, 311
Global variables
- declaring, 26
- functions, 122–124

gmtime function, 340–341
Golden ratio app, 20–23
good function, 383
Gordon, Peter, 18
goto statements
- description, 93
- overview, 105–106

[:graph:] character class, 623
greater function-object template, 563
greater_equal function-object template, 563
Greater than signs (>)
- comparisons, 12, 70–71
- shift operators, 8, 70, 76, 81
- stream-input operator, 4

Greenwich mean time function, 341
Groups in regular expressions, 614, 622–623
Guess-the-Number game app, 111–113
Guess-the-Word game app, 543–546

H

h format specifier
- printf, 316
- scanf, 320

%H format specifier for date and time, 345
Hash functions
- hash-table performance, 538–541
- unordered containers, 537–538
- writing, 541–543

Header files, 263–264
Heap, 65
- algorithms, 560–561
- priority queues, 488

hex manipulator, 370, 372
Hexadecimal characters
- format flags, 378
- format specifiers, 314, 316, 319
- literals, 32
- regular expressions, 619
- special character, 44
- stream manipulator, 372
- testing for, 303

Hidden "this" pointers, 217–218
Hierarchies in name conflicts, 239–240
High-precedence operators, 62–64
Horizontal tab character, 43
Hyperbolic functions, 335–336

I

%I format specifier for date and time, 345
%i format specifier for printf, 313
Idiot savant app, 358–360
if and if-else statements
- description, 92
- overview, 96–98
- working with, 11–13

#if preprocessor directive
- conditional compilation, 18, 257–259
- description, 252

#ifdef preprocessor directive
- conditional compilation, 258
- description, 252

#ifndef preprocessor directive
- conditional compilation, 258
- description, 252

ifstream class, 366–367, 387, 390
ignore function, 380
Implementation of classes, 194
in flag, 388, 395
#include preprocessor directive, 2, 249, 252–253
includes algorithm
- detailed description, 573
- purpose, 557

Inclusive OR operators
- assignment, 76
- detail, 78–79

Incoming conversion functions, 228–229
Increment operators
- class support, 227–228
- description, 13
- postfix, 62–63
- prefix, 64

Index operator ([])
- arrays, 163
- pointers, 175–178

Indexes
    arrays, 155, 163, 165
    deques, 448
    strings, 408–410
    subscript operator, 225–226
Indirection (*) operator, 160–165, 175–178
Inequality operator (!=) for comparisons, 12
Inheritance of classes. *See* Deriving classes
Inherited constructors, 232–233
Initialization
    default members, 215–216
    strings, 293
    variable declarations, 26
Initialization lists
    arrays, 153–154
    constructors, 211–214
Inline functions, 124
    calls to, 196
    class templates with, 276
    inline modifier, 125–126
inner_product algorithm
    default behavior, 562
    detailed description, 574
inplace_merge algorithm
    detailed description, 574
    purpose, 558
Input
    iterators, 552
    with right-shift operator, 363–364
    stream functions, 379–381
    strings, 293–294
insert function
    deques, 454–455
    lists, 474–475
    maps, 515–516
    multisets, 533
    sets, 520, 526–527
    strings, 419–420, 429
    vectors, 435, 443–444
Insert iterators, 553–554
inserter function, 553–554
Inserting
    set elements, 519–520
    string contents, 419–420
Instantiating templates, 266, 274–275
int and integer data types
    vs. bool, 34–35
    declaring, 8, 26
    description, 27
    vs. floating point, 34
    format flags, 378
    format specifiers, 313, 318
    guidelines, 28–29
    limitations, 21
    literals, 31
    template parameters, 278–279
Interfaces, 238
internal manipulator, 372
invalid_argument exception, 108
Inverse tangent function, 335
I/O functions, 309
    console, 310–313
    files, 321–331
    overview, 309–310
    strings, 321
I/O stream classes, 361
    basics, 361–362
    file. *See* File stream operations
    format flags, 377–378
    hierarchy, 366–368
    input with >>, 363–364
    manipulators, 368–369, 371–378
    member functions, 379–385
    output with <<, 362
    reading and writing string streams, 395–397
    reading lines with getline, 364–366
    shift operator overloading, 398–399
    text file reader app, 400–401
<iomanip> header, 368
ios_base flags, 378, 387–388
<iostream> header, 361, 368
iota algorithm
    default behavior, 562
    detailed description, 574–575
is_heap algorithm
    detailed description, 575
    purpose, 561
is_heap_until algorithm
    detailed description, 575
    purpose, 561
is_open function, 391
is_partitioned algorithm
    detailed description, 575–576
    purpose, 559

is_permutation algorithm
    detailed description, 576
    purpose, 559
is_sorted algorithm
    detailed description, 576
    purpose, 559
is_sorted_until algorithm
    detailed description, 576
    purpose, 559
isalnum function, 302
isalpha function, 302
iscntrl function, 302
isdigit function, 302
isgraph function, 302
islower function, 303
isprint function, 303
ispunct function, 303
isspace function, 303
istream class, 70
    hierarchy, 366–367
    manipulators, 369
isupper function, 303
isxdigit function, 303
iter_swap algorithm
    detailed description, 577
    purpose, 558
Iterating maps, 506–507
Iterative searches, 626–628
Iterators
    algorithms, 551–552
    arithmetic, 427–428, 436
    deques, 449–450
    forward, 424–426
    introduction, 433–434
    lists, 467–469
    member functions using, 428–429
    reverse, 426–427
    sets, 520–521
    strings, 424–427
    unordered containers, 535
    vectors, 436–438

J

%j format specifiers, 345
Join operator, 58, 76–77
Joining strings, 69, 293, 295, 297, 407
Jump statements, 104–106

K

Keys
    associated containers, 432
    map template, 497–500
    unordered containers, 535, 537–538
Keywords, new, 651–652
knuth_b engine, 605

L

l format specifier
    printf, 316
    scanf, 320
L format specifiers, 317
Labels
    description, 93
    goto statements, 105–106
labs function, 357
Lambda functions, 131
    basic syntax, 117, 132–133
    closure syntax, 133–136
    mutable keyword, 136–137
    storing and returning, 138–141
    working with, 137–138, 550–551
Last-in-first-out (LIFO) storage, 481–484
Lavavej, Stephan T., 647
ldexp function, 81, 337–338
ldiv function, 357–358
left manipulator, 372
Left-shift operators (<<), 76
    bitwise, 81
    output with, 8, 362
    working with, 69–70
Length
    arrays, 153
    strings, 297, 420
length_error exception, 108
length function, 420
less function-object template, 563
less_equal function-object template, 563
Less than signs (<)
    comparisons, 12, 70–71
    maps, 509
    shift operators, 8, 69–70, 76, 81
Letters, testing for, 302
Lexical analysis, 300, 613, 633
lexicographical_compare algorithm, 577
Line comments, 17

Line-continuation characters, 249
__LINE__ predefined macro, 262
#line preprocessor directive, 253
Lines, reading, 364–366
Lists and list templates, 465
  constructors, 469–470
  description, 433
  functions, 471–481
  iterators, 467–469
  overview, 466–467
  sorting, 465–466
  vectors, 439
Literals, 25
  numeric, 31–33
  string. *See* String and character literals
Little-endian systems, 242
Load exponent functions, 337–338
Load factor
  hash-table performance, 538–539
  unordered containers, 540
load_factor function, 540
Local variables with functions, 122–124
localtime function, 340–341
log function, 338
log output object, 368
log10 function, 338
Logarithm functions, 338
logic_error class, 107
logical_and function-object template, 564
logical_not function-object template, 563
Logical operators
  AND, 72–73
  negation, 64
  OR, 72–73
logical_or function-object template, 564
Long numbers
  format specifiers, 316–317, 320
  long data type, 27, 31
  long double data type, 28
  long long data type, 28, 31
Loops
  arrays, 154–155
  for statements, 99–102
  while and do-while statements, 13–14, 98–99
lower_bound function
  detailed description, 577
  maps, 516
  multimaps, 531
  multisets, 533
  purpose, 557
  sets, 528
[:lower:] character class, 623
Lowercase characters
  converting to, 304
  testing for, 303
Lvalues
  references. *See* References
  and rvalues, 646–647
  working with, 59

M

%m format specifiers, 345
%M format specifiers, 346
Macros
  #define for, 256–257
  predefined, 260–262
Magic numbers, 254
main function, 2–4
make_heap algorithm
  detailed description, 577–578
  purpose, 561
make_pair function, 496
make_tuple function, 286
malloc function, 349–350
Manipulators
  descriptions, 371–378
  overview, 368–369
Mantissas
  format specifiers, 313
  functions for, 337
<map> header, 509, 530
Maps and map templates, 495
  binary trees, 507–509
  constructors, 509–511
  description, 433
  finding elements, 501–502
  functions, 511–518
  iterating through, 506–507
  multimap template, 529–532
  overview, 497–498
  pair template, 495–497
  populating, 498–500
  record type, 502–506

Matching and searching functions, 624–626
Math functions, 336–339
<math.h> header, 334
max algorithm
  detailed description, 578
  purpose, 557
max_element algorithm
  detailed description, 578
  purpose, 557
max function, 610
max_load_factor function, 540
max_size function
  deques, 456
  lists, 475
  strings, 420
  vectors, 444
Median app, 491–493, 555–556
Member functions
  calling, 198–200
  class templates with, 276–278
  defining, 195–198
  file-specific, 390–391
  I/O stream classes, 379–385
  iterators, 428–429
  overriding, 238–239
  private, 200–201
  string class, 410–424
Members
  accessing, 63
  initialization defaults, 215–216
  operator functions as, 219–221
  static, 203–205
memccpy function, 304–305
memchr function, 305
memcmp function, 305
memcpy function, 305
memmove function, 306
Memory
  byte searches in, 305
  comparing, 305
  copying, 304–305
  leaks, 66, 180–181
  moving, 306
  pointers. *See* Pointers
  setting, 306
Memory allocation
  functions, 348–350
  pointers, 167–169
Memory-block functions, 304–306
memset function, 306
merge function
  detailed description, 578
  lists, 475–476
  purpose, 558
mersenne_twister engine, 604
Microsoft Visual Studio, 5
min algorithm
  detailed description, 578
  purpose, 557
min_element algorithm
  detailed description, 579
  purpose, 557
min function, 609–610
minmax_element algorithm
  detailed description, 579
  purpose, 557
minstd_rand engine, 604
minstd_rand0 engine, 604
minus function-object template, 563
Minus signs (-)
  arithmetic negation, 65
  assignment operator, 74–75
  decrement operator, 63–64
  format specifiers, 316
  regular expressions, 616, 618–619
  subtraction, 10, 69
mismatch algorithm, 579
Mixing numeric types, 33–38
mktime function, 342–343, 358
mode specifiers, 323–324
modf function, 338
Modifiers
  function declarations, 124–126
  variable declarations, 47–50
Modifying algorithms, 556–558
modulus function-object template, 563
Modulus operators, 68
  assignment, 76
  fmod, 337
  random numbers, 599–600
move algorithm
  detailed description, 580
  purpose, 558
move_backward algorithm
  detailed description, 580
  purpose, 558

Move constructors, 644
Move semantics, 637, 640–642
Moving memory, 306
mt19937 engine, 604
mt19937_64 engine, 605
Multidimensional arrays, 157–159
multimap template, 529–532
Multiple line comments, 18
Multiplication
    assignment operator, 75
    operator, 10, 68
    precedence, 58
multiplies function-object template, 563
multiset template, 532–534
mutable keyword, 136–137

N

\n characters
    regular expressions, 619
    special character, 43
Named unions, 242–244
Names
    conflicts, 239–240
    files, 330
    function variables, 122–124
    symbolic, 30
Namespaces, 15–17
Natural logarithm function, 338
NDEBUG predefined macro, 261
negate function-object template, 563
Negation operators, 64–65
Nesting anonymous unions, 245
New features, 649
    core-language enhancements, 650–651
    keywords, 651–652
    object construction, 649–650
    standard library, 652–653
new keyword
    memory-allocation, 348–349
    pointers, 168–169
    primitive types, 65–66
Newlines
    regular expressions, 619
    special character, 43
    stream manipulator, 371
next_permutation algorithm
    detailed description, 581
    purpose, 559
noboolalpha manipulator, 372
none function for bitsets, 460
none_of algorithm
    detailed description, 581
    purpose, 557
normal_distribution distribution, 607
noshowbase manipulator, 373
noshowpoint manipulator, 373
noshowpos manipulator, 373
noskipws manipulator, 373
nounitbuf manipulator, 373
nouppercase manipulator, 373–374
nth_element algorithm, 582
Null statements, 91, 93–94
nullptr keyword, 601
Nulls and NULL values
    pointers, 169
    printing, 371
    strings, 40, 291–292
Number guessing apps, 111–116
Number signs (#)
    format specifiers, 316
    preprocessor directives, 250
<numeric> header, 547, 562
Numeric types
    algorithms, 562
    literals, 31–33
    mixing, 33–38
    special characters, 44
    string-to-number conversions, 347–348

O

%o format specifiers
    printf, 314
    scanf, 318
object.member syntax, 63, 190, 192
Objects
    child, 233–235
    creating and using, 199–200
    deleting, 66
    destructors, 216–217
    file, 387–390

  function, 224–225, 562–564
  new features, 649–650
  overview, 189–190
  this pointer, 217–218
oct manipulator, 374
Octal format
  format flags, 378
  format specifiers, 314, 316, 318
  literals, 32
  special character, 43
  stream manipulator, 374
Odds at dice app, 142–145
Offsets in random-access operations, 393–394
ofstream class, 366, 387, 390
One-dimensional arrays, 153–154
open function, 391
Opening files, 309, 322–324, 391
Operator functions
  assignment, 222–224
  as friends, 221–222
  function-call, 224–225
  increment and decrement, 227–228
  as members, 219–221
  overview, 218–219
  subscript, 225–226
Operators, 57
  add and subtract, 69
  arithmetic, 10
  assignment, 73–76, 222–224
  associativity, 58
  binary printout app, 88–90
  bitwise, 72–73, 78–81
  cast, 82–88
  comparison, 12
  conditional, 73
  data-access, 62–64
  join, 76–77
  less than and greater than, 70–71
  logical conjunctions, 72–73
  lvalues, 59
  multiply and divide, 68
  overloading, 218
  pointer-to-member, 67–68
  postfix, 77–78
  precedence, 57–58
  prefix, 64–66, 77–78
  preprocessor directives, 259–260
  scope, 62
  shift, 69–70, 398–399
  string class, 406–410
  summary, 59–61
  test for equality, 71–72
  throw, 76
OR operators
  assignment, 76
  description, 72
  detail, 78–79
  regular expressions, 616
ostream class, 70
  hierarchy, 366–367
  manipulators, 369
out flag, 388, 395
out_of_range exception, 108
Outgoing conversion functions, 228–229
Output
  iterators, 552
  left-shift operator, 362
  stream functions, 381–382
  strings, 293–294
overflow_error exception, 108
Overloading
  constructors, 208–210
  functions, 126–128
  operators, 218
  shift operators, 398–399
override keyword
  deriving classes, 238–239
  function declarations, 126
Overriding subclasses, 235–237

P

%p format specifier
  date and time, 346
  printf, 315
Packed Boolean app, 245–248
pair template, 495–497
Parameters with function templates, 267–272
Parentheses ()
  function calls, 121, 224–225
  operator precedence, 58
  regular expressions, 614

partial_sort algorithm
  detailed description, 582
  purpose, 559
partial_sort_copy algorithm
  detailed description, 582–583
  purpose, 559
partial_sum algorithm
  detailed description, 583
  purpose, 562
partition algorithm, 583–584
partition_point algorithm
  detailed description, 584
  purpose, 559
Passing
  arrays to functions, 155–156
  function arguments, 121
  multidimensional arrays, 158
  pointers, 178–180
Patterns in regular expressions. *See* Regular expressions
PAUSE command, 4–5
peek function, 381
Percent signs (%)
  modulus assignment operator, 68, 76
  printf format specifiers, 313–315
  scanf format specifiers, 317–319
Phi calculation app, 20–23
piecewise_constant_distribution distribution, 608
piecewise_linear_distribution distribution, 608
plus function-object template, 563
Plus signs (+)
  addition, 10, 69
  assignment operator, 74–75
  format specifiers, 315
  increment operators, 13, 62–64
  operator precedence, 58
  pattern modifiers, 621
  regular expressions, 614
  strings, 407
Pointer-to-break function, 298
Pointer-to-member operators, 67–68
Pointers, 159
  as arguments, 162–163
  arithmetic, 166–167
  array access, 163–166
  vs. array arguments, 167
  base-class, 233–235
  concepts, 159–161
  const for, 172–173
  to const types, 170–172
  declarations with, 175–178
  dereference operator, 65
  format specifiers, 315
  function, 178–180
  memory allocation, 167–169
  smart, 180–186
  void, 173–175
poisson_distribution distribution, 607–608
Polymorphism, 189
pop function
  priority_queues, 491
  queues, 485, 487
  stacks, 482–484
pop_back function
  deques, 456
  lists, 476
  vectors, 444
pop_front function
  deques, 448, 456
  lists, 476
pop_heap algorithm
  detailed description, 584
  purpose, 561
Populating
  maps, 498–500
  sets, 519–520
position function for regular expressions, 625–626
Position in random-access files, 328–329
Postfix operators
  precedence, 62–63
  vs. prefix, 77–78
Pound signs (#)
  format specifiers, 316
  preprocessor directives, 250
pow function, 338
Power function, 336, 338
#pragma preprocessor directive, 253
Precedence
  high-precedence operators, 62–64
  operators, 57–58
  pointers, 163–164
Precision
  floating-point numbers, 381
  format specifiers, 316–317
  stream manipulator, 375
precision function, 381

Predefined function objects, 562–564
Predefined macros, 260–262
Prefix operators
  vs. postfix, 77–78
  precedence, 64–66
Preprocessor directives, 249
  conditional compilation, 257–259
  creating macros, 256–257
  general syntax, 249–250
  meaningful symbols, 254–255
  operators, 259–260
  predefined macros, 260–262
  project header files, 263–264
  summary, 250–253
prev_permutation algorithm
  detailed description, 584–585
  purpose, 559
Prime numbers app, 186–188
Primitive data types, 25
  floating-point, 29–30
  integers, 28–29
  overview, 27–28
[:print:] character class, 623
Print fields
  fill characters, 375, 381
  format specifiers, 317
  justification, 316, 372, 374, 378
  minimum width, 376, 382
  padded, 316
  stream manipulators, 370, 376
Printable characters, testing for, 302–303
printf function, 309–310
  description, 311
  format specifiers, 313–315
priority_queue template, 465
  constructors, 489–490
  functions, 490–491
  overview, 487–489
private keyword
  base-class specifiers, 231–232
  data-member access, 194
  derived classes, 230
  member functions, 200–201
Procedures. *See* Functions
Project header files, 263–264
protected keyword
  base-class specifiers, 232
  data-member access, 194
  derived classes, 230
  member functions, 201
Prototypes, function, 117–120
Pseudorandom numbers, 350–351, 600
ptr operator, 63
public keyword
  base-class specifiers, 231
  data-member access, 193–194
  derived classes, 230
[:punct:] character class, 624
Punctuation characters, testing for, 303
Pure virtual functions, 238
push function
  priority_queues, 490–491
  queues, 487
  stacks, 481–484
push_back function
  deques, 450, 456
  insert iterators, 554
  lists, 469, 477
  strings, 420
  vectors, 435, 444–445
push_front function
  deques, 448, 456
  insert iterators, 554
  lists, 477
push_heap algorithm
  detailed description, 585
  purpose, 561
put function, 381
putback function, 381
putc function, 327
putchar function, 311
puts function, 309–311, 325

Q

qsort function, 173–174
  declaration, 179
  support, 351
  working with, 353–355
Question marks (?)
  conditional operator, 73
  pattern modifiers, 621
  regular expressions, 616
<queue> header, 485, 488
Queue template, 484–487
Queues, double-ended. *See* deque templates

Quick sort. *See* qsort function
Quote signs (',")
    double-quoted strings, 40–41
    single-quoted characters, 39–40, 295
    special character, 43
    stringizing operator, 259
    strings, 6, 40–41, 293
Quoted strings, #define with, 254–255

R

\r characters
    regular expressions, 619
    special character, 43
r mode specifiers, 323
r+ mode specifiers, 323
Radian measures, 334
rand function, 111–112, 350–351, 599–600
Random-access operations
    file stream operations, 392–394
    functions, 328–329
    iterators, 552
<random> header, 601, 605
random_shuffle algorithm
    detailed description, 585–586
    purpose, 559
    randomization schemes, 603
Randomization functions, 350–351
Randomization library, 599
    biased distributions, 599–600
    common engines, 604–605
    dice game app, 610–612
    distribution operations, 609–610
    distributions, 605–608
    engine operations, 608–609
    issues summary, 599
    pseudorandom sequences, 600
    randomization schemes, 601–603
    seeds, 600–601
Range-based for statements
    arrays, 155–156
    overview, 101–102
range_error exception, 108
Ranges
    bit fields, 241
    regular expressions, 614, 618–619
ranlux24_base engine, 604
Raw string literals, 33, 46
rbegin function
    deques, 456–457
    lists, 469, 477
    maps, 516–517
    sets, 528
    strings, 426–427
    vectors, 436–437, 445
read function in binary mode, 391
Read-only algorithms, 556–557
Read-only mode, 323
Read/write mode, 323
Readability, #define for, 255
Reading
    binary files, 327–328
    in binary mode, 391–392
    console input, 312
    input with >>, 363–364
    lines with getline, 364–366
    string streams, 395–397
    text files, 325–327
realloc function, 348–350
Record type in map template, 502–506
Recurring groups in regular expressions, 622–623
Recursive statements, 91
Reference counts for pointers, 181
Reference operator (&)
    copy constructors, 210
    pointers, 159, 163, 175
References, 147
    arguments, 121, 147–150
    const modifier, 151–152
    dangling, 140
    returning from functions, 150–151
regex_error class, 631
<regex> header, 613, 624, 631
regex_iterator function, 633
regex_iterator template, 627
regex_match function, 615, 624
regex_replace function, 628–629
regex_search function, 615, 624–626
regex_token_iterator function, 633
register modifier, 48
Regular expressions, 613
    character classes, 623–624

- characters, 618–620
- escape sequences, 614, 616–618
- exceptions, 631–632
- iterative searches, 626–628
- matching and searching functions, 624–626
- overview, 613–616
- pattern modifiers, 620–622
- recurring groups, 622–623
- replacing text, 628–629
- RPN calculator app, 632–635
- string tokenizing, 630–631

rehash function, 539–540
Rehashes in hash-table performance, 539
reinterpret_cast operator
- binary files, 327
- binary mode, 391
- memory access, 348
- named unions, 243
- overview, 84–85
- pointers, 174–175
- searching and sorting functions, 351, 353

Remainder operations, 68
- assignment, 76
- fmod, 337
- random numbers, 599–600

remove algorithm
- detailed description, 586
- purpose, 558

remove_copy algorithm
- detailed description, 586–587
- purpose, 558

remove_copy_if algorithm
- detailed description, 587–588
- purpose, 558

remove function
- files, 330
- lists, 477

remove_if algorithm
- detailed description, 588
- lists, 478
- purpose, 558

rename function, 330
Renaming files, 330
rend function
- deques, 457
- lists, 469, 478
- maps, 517
- sets, 528
- strings, 426–427
- vectors, 437, 445

Reordering algorithms, 558–559
replace algorithm
- detailed description, 588
- purpose, 558

replace_copy algorithm
- detailed description, 588–589
- purpose, 558

replace_copy_if algorithm, 589
replace function
- string iterators, 429
- strings, 420–421

replace_if algorithm
- detailed description, 589
- purpose, 558

Replacing text
- regular expressions, 628–629
- strings, 420–421

reserve function
- strings, 421–422
- unordered containers, 541
- vectors, 445–446

reset function
- bitsets, 460
- shared_ptr, 182–183
- unique_ptr, 185

resetiosflags manipulator, 374, 377
resize function
- deques, 457
- lists, 478
- strings, 422
- vectors, 446

Return values and return statement, 4
- constexpr functions, 141
- description, 93
- function definitions, 120
- function prototypes, 119
- lambda functions, 132, 138–141
- pointers, 178–180
- references from function, 150–151

reverse algorithm
- detailed description, 589–590
- purpose, 559

reverse_copy algorithm
    detailed description, 590
    purpose, 559
Reverse finds for strings, 422–423
reverse function, 478
reverse_iterator function
    deques, 450
    lists, 469
Reverse iterators
    strings, 426–427
    vector templates, 437–438
Reverse Polish Notation (RPN) calculator app, 632–635
Reversing
    lists, 478
    strings, 298
rewind function, 331
rfind function, 422–423
right manipulator, 374
Right-shift operators (>>)
    bitwise, 81
    input with, 363–364
    working with, 70
rotate algorithm
    detailed description, 590
    purpose, 559
rotate_copy algorithm
    detailed description, 590–591
    purpose, 559
Rounding functions, 336–337
RPN (Reverse Polish Notation) calculator app, 632–635
runtime_error class, 107
Runtime-performance, 645–646
Rvalue references, 637
    contained objects, 646
    in copying, 637–640
    and lvalues, 646–647
    in move semantics, 640–642
    runtime-performance, 645–646
    in string class, 642–644

S

\s characters in regular expressions, 620
\S characters in regular expressions, 620
%s format specifiers
    printf, 314
    scanf, 319
%S format specifier for date and time, 346
Scaling arrays, 156
scanf function, 309
    description, 312
    format specifiers, 317–320
Scientific format
    format flags, 378
    stream manipulator, 374
scientific manipulator, 374
Scientific notation
    format specifiers, 313
    literals, 32
Scope and scope operator (::)
    copy constructors, 210
    default constructors, 207
    enum type, 52
    function variables, 122–124
    member functions, 195–196
    name conflicts, 239–240
    working with, 62
search algorithm
    detailed description, 591
    purpose, 557
search_n algorithm
    detailed description, 591
    purpose, 557
Searching functions, 351
    bsearch, 352–353
    regular expressions, 624–626
seed function, 609
Seeds
    engines, 609
    random numbers, 351, 600–601
seekg function, 394
seekp function, 394
Semicolons (;)
    class declarations, 191
    function definitions, 120, 124
    lambda functions, 133
    null statements, 91, 93–94
    after statement terminators, 6, 9
Sequence containers, 432
set_difference algorithm
    detailed description, 591–592
    purpose, 559
set function for bitsets, 460–461
<set> header, 518

set_intersection algorithm
  detailed description, 592
  purpose, 559
set_symmetric_difference algorithm
  detailed description, 592
  purpose, 559
set templates
  constructors, 521–523
  description, 433
  finding set elements, 520–521
  functions, 523–529
  overview, 518–519
  populating sets, 519–520
set_union algorithm
  detailed description, 593
  purpose, 559
setf function, 383–384
setfill manipulator, 375
setiosflags manipulator, 375, 377
setprecision manipulator, 375
Setting memory, 306
setw manipulator, 376
shared_ptr pointer, 180–184
Shift operators, 8
  assignment, 76
  bitwise, 81
  input with, 363–364
  output with, 8, 362
  overloading, 398–399
  working with, 69–70
Short numbers
  data types, 27
  format specifiers, 316, 320
showbase manipulator, 370, 376
showpoint manipulator, 376
showpos manipulator, 376
Sieve of Eratosthenes app, 186–188
Sign-extended bit patterns, 80
Signed types
  bitwise operators, 79–81
  vs. unsigned, 35–38
Simple arrays, 153–154
Simple variables, 25–26
sin function, 335
Single quotation marks (')
  single-quoted characters, 39–40, 295
  special character, 43
sinh function, 335
Size and size function
  bitsets, 461
  deques, 448, 456–457
  lists, 475, 478–479
  maps, 517
  priority_queues, 491
  queues, 487
  sets, 528
  stacks, 483
  strings, 420, 422–423
  vectors, 446
sizeof operator
  description, 65
  variadic templates, 286
skipws manipulator, 376
Slashes (/)
  assignment operator, 76
  comments, 8, 17–18
  division, 10, 68
Smart pointers
  overview, 180–181
  shared_ptr, 180–184
  unique_ptr, 184–186
smatch type, 625
sort algorithm
  detailed description, 593
  purpose, 559
  syntax, 548
sort function, 479
sort_heap algorithm
  detailed description, 593–594
  purpose, 561
Sorting
  algorithms, 558–559
  functions, 351–355
  list elements, 465–466, 479
[:space:] character class, 624
Spaces in format specifiers, 316
Spaghetti code, 106
Special characters, 41–45
Special declarations, 52–54
Specialization, template, 279–281
Spectral characteristics of random numbers, 600
splice function, 479–481
sprintf function, 321
sqrt function, 338–339

Square brackets ([])
  arrays, 153, 163, 225–226
  bitsets, 459
  C-strings, 293
  lambda functions, 132–135, 550–551
  multidimensional arrays, 157–159
  pointers, 175–178
  records, 503
  regular expressions, 614, 618–619
  strings, 408–410
srand function, 111–112, 350–351
sregex_iterator type, 626–627
sregex_token_iterator type, 630
sscanf function, 321
stable_partition algorithm, 594
stable_sort algorithm
  detailed description, 594–595
  purpose, 559
<stack> header, 481
Stack template, 481–484
static_assert macro, 261
static_cast operator, 34, 83–84
static keyword and static members
  classes, 49, 203–205
  function declarations, 126
  function variables, 124
  functions, 124
std: prefix, 2–3
  exceptions, 107–108
  I/O stream classes, 361
  namespaces, 15–16
stdafx.h file, 5
__STDC__ predefined macro, 262
stdin file pointer, 309, 325
<stdio.h> header, 310, 322
<stdlib.h> header, 347, 349, 351
stdout file pointer, 309, 325
Storage classes, 123–124
Storing lambdas, 138–141
str function
  regular expressions, 625–626
  versions, 396–397
strcat function, 293, 295
strchr function, 295–296
strcmp function, 296
strcoll function, 296
strcpy function, 293, 296
strcspn function, 296–297
Stream classes. *See* I/O stream classes
Stream-input operator (>>), 4, 294
Stream manipulators
  descriptions, 371–378
  overview, 368–369
Stream-output operator (<<), 293
streamoff type, 394
streamsize type, 392
strerror function, 297
strftime function, 339, 342, 344–345
Strict weak ordering, 465–466
String and character literals, 39
  double-quoted strings, 40–41
  raw, 46
  single-quoted characters, 39–40
  special characters, 41–45
  wide-character strings, 45–46
string class, 41, 403
  constructors, 405–406
  iterators, 424–427
  member functions, 410–424
  operators, 406–410
  overview, 403–404
  rvalue references in, 642–644
  wide-character, 430
<string> header, 403–404
<string.h> header, 294
Stringizing operator, 259
Strings, 6
  assigning, 406
  C-string, 291–299
  character functions, 301–304
  comparing, 296–297, 408, 413
  concatenating, 69, 293, 295, 297, 407
  conversion functions, 347–348
  copying, 296–297
  error, 297
  format specifiers, 314
  indexing, 408–410
  I/O functions, 321
  length, 297, 420
  library functions, 291
  literals. *See* String and character literals
  memory-block functions, 304–306
  pointer-to-break, 298
  reading and writing, 395–397

  reversing, 298
  substrings, 295–296, 298–299
  tokenizing, 299–301, 630–631
  transforming, 299
  wide-character, 45–46, 306–307, 430
stringstream class, 395–396
strlen function, 297
strncat function, 293, 297
strncmp function, 297
strncpy function, 297
stroul function, 348
strpbrk function, 298
strrchr function, 298
strspn function, 298
strstr function, 298–299
strtod function, 347–348
strtok function, 299–301, 414, 633
strtol function, 348
struct keyword, 190–191, 198
Structures
  creating, 198
  declaring, 191
  overview, 189–190
strxfrm function, 299
student_t_distribution distribution, 608
Subclassing. *See* Deriving classes
Subroutines. *See* Functions
Subscript operator ([])
  arrays, 153, 157–159, 163, 225–226
  bitsets, 459
  C-strings, 293
  records, 503
  strings, 408–410
substr function, 423
Substrings
  finding, 295–296, 298–299, 415–416
  retrieving, 423
Subtraction
  assignment operator, 74–75
  example, 10
  overview, 69
swap function
  deques, 457–458
  detailed description, 595
  lists, 481
  maps, 517
  purpose, 558
  sets, 529
  strings, 423–424
  vectors, 447
swap_ranges algorithm
  detailed description, 595
  purpose, 558
switch statements
  break statements, 104–106
  description, 92
  overview, 103–104
Symbols
  #define for, 254–255
  enumerated types, 51
  rules, 30
sync_with_stdio function, 382, 385
system function, 358

T

\t characters
  regular expressions, 619
  special character, 43
t mode specifiers, 323
Tab character in regular expressions, 619
Tagged characters in regular expressions, 622
tan function, 335–336
tangent functions, 334–336
tanh function, 336
Target strings in regular expressions, 614
tellg function, 394
tellp function, 394
tempfile function, 331
Templates, 265
  bitset, 458–461
  classes, 272–275
  containers, 431–433
  deque. *See* deque templates
  function, 267–272
  integer parameters, 278–279
  list. *See* Lists and list templates
  map. *See* Maps and map templates
  median app, 491–493
  multimap, 529–532
  multiset, 532–534
  pair, 495–497
  priority_queue, 487–491
  queue, 484–487
  set. *See* set templates
  specialization, 279–281

Templates (*continued*)
  stack, 481–484
  syntax and overview, 265–266
  type promotion app, 288–289
  variadic. *See* Variadic templates
  vector. *See* Vectors and vector templates
tempname function, 331
Terminating null bytes, 291
Terminating programs, 356–357
Test for equality operators, 71–72
test function for bitsets, 461
Text. *See also* Strings
  literals, 33
  preprocessor replacement, 250
  regular expressions, 628–629
Text files
  vs. binary, 385–386
  reader app, 400–401
  reading and writing, 325–327
Text mode, 323
this pointer
  assignment operator, 223
  hidden, 217–218
thread_local modifier, 49
throw operator, 58, 76
Throwing exceptions, 110–111
Tildes (~) for bitwise negation, 64
Time and date functions, 339
  format specifiers, 344–346
  overview, 340–342
  random number seeds, 603
  tm structure, 342–344
time function, 339, 342, 603
__TIME__ predefined macro, 262
<time.h> header, 339–340
time_t type, 339
tm structure, 339–340, 342–344
to_string function, 458, 461
to_ullong function, 461
to_ulong function, 461
Token-compression operator, 260
Tokenizing strings, 299–301, 630–631
tolower function, 304
top function
  priority_queues, 491
  stacks, 482–484
toupper function, 40, 304
transform algorithm
  detailed description, 595–596
  purpose, 558
Transforming strings, 299
Trees, binary
  heap algorithms, 560–561
  maps, 507–509
Trigonometric functions, 333–336
true value, 32
trunc mode flag, 388
try-catch blocks
  exception objects, 109–110
  regular expressions, 631–632
  syntax, 108–109
tuple_element class, 287
<tuple> header, 286
tuple_size class, 287
Tuples, 286–287
Two's complement format, 36–38
Type promotion apps, 54–55, 288–289
typedef keyword
  declarations, 52–53
  pointers, 178–179
typeid operator, 54, 63–64
typename keyword, 265–266
  class templates, 278–279
  variadic templates, 285
Types. *See* Data

U

\u characters in regular expressions, 619
%u format specifiers
  printf, 314
  scanf, 319
%U format specifiers for weeks, 346
ulong function, 459
#undef preprocessor directive, 253
underflow_error exception, 108
Underscores (_) in symbolic names, 30
unget function, 312
ungetc function, 327
Unicode characters in regular expressions, 619
uniform_int_distribution distribution, 602, 605
uniform_real_distribution distribution, 606

union keyword, 190
Unions, 242
  anonymous, 244–245
  named, 242–244
  overview, 189–190
unique_copy algorithm
  detailed description, 597
  purpose, 559
unique function
  detailed description, 596–597
  purpose, 559
  shared_ptr, 184
unique_ptr pointers, 180–181, 184–186
unitbuf manipulator, 376
Unordered containers, 534
  basic concepts, 535–538
  Guess-the-Word app, 543–546
  hash-table performance, 538–541
  writing hash and equals functions, 541–543
unsetf function, 384–385
Unsigned types
  bit fields, 240
  bitwise operators, 79–81
  char, 27
  format specifiers, 314
  int, 27
  literals, 31
  long, 27
  long long, 28
  short, 27
  vs. signed, 35–38
Up-casting, 233–235
upper_bound function
  detailed description, 597
  maps, 517–518
  multimaps, 531
  multisets, 533
  purpose, 557
  sets, 529
[:upper:] character class, 624
uppercase characters
  converting to, 304
  testing for, 303
uppercase manipulator, 377
use_count function, 184
User-defined types, 25
using statement, 2–3, 7
  namespaces, 15–17
  string class, 404

V

\v characters
  regular expressions, 619
  special character, 43
va_arg function, 130–131
va_end function, 130
va_list function, 130–131
va_start function, 130–131
Value, passing arguments by, 121
Variable-length argument lists, 129–131
Variables
  declarations, 7–10
  functions, 122–124
  global, 26
  lambda functions, 550–551
  modifiers, 47–50
  pointers, 159–161
  simple, 25–26
Variadic templates
  overview, 281–283
  rules summary, 285–286
  sophisticated, 284
  tuples, 286–287
Vectors and vector templates
  constructors, 438–439
  description, 433
  functions, 439–447
  heap algorithms, 560–561
  iterators, 436–438
  list initialization, 439
  overview, 434–436
Vertical bars (|)
  inclusive OR operator, 76
  logical OR operator, 73
  pattern modifiers, 621
  regular expressions, 616
Vertical tabs
  regular expressions, 619
  special character, 43
Virtual functions
  pure, 238
  subclasses, 235–237
virtual modifier in function declarations, 126

void pointers, 173–175
void return type, 119–120, 132
volatile modifier, 49–50

W

\w characters in regular expressions, 620
\W characters in regular expressions, 620
%w format specifiers, 346
%W format specifiers, 346
w mode specifiers, 323
w+ mode specifiers, 323
w_char data type, 28
<wchar.h> header, 306
wchar_t type, 45–46, 306–307, 430
weak_ptr pointer, 180–181
weibull_distribution distribution, 608
what function, 631
while statements
    break statements, 104–106
    description, 92
    loops, 13–14
    overview, 98–99
Whitespace
    regular expressions, 620
    scanf, 317
    stream manipulator, 376
    testing for, 303
Wide-character strings, 430
    functions, 306–307
    literals, 33
    working with, 45–46
Width
    bit fields, 241
    format specifiers, 316–317, 320
    print fields, 382
width function, 382
wmatch type, 625
Word boundaries in regular expressions, 620
wprintf function, 312–313
write function, 392
Writing
    binary files, 327–328
    in binary mode, 391–392
    to files, 386
    output with <<, 362
    string streams, 395–397
    text files, 325–327
ws manipulator, 377
wscanf function, 312

X

\x characters in regular expressions, 619
%x format specifier
    date, 346
    printf, 314
    scanf, 319
%X format specifier
    printf, 314
    scanf, 319
    time, 346
[:xdigit:] character class, 624

Y

%y format specifiers, 346
%Y format specifiers, 346

Z

%Z format specifiers, 346
Zero-based arrays, 152, 155
Zeros in format specifiers, 316