

Siddhartha Rao

SEVENTH
EDITION

Covers
C++11

Sams **Teach Yourself**

C++

in **One Hour** a Day

SAMS

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

Siddhartha Rao

Sams **Teach Yourself**

C++

in **One Hour** a Day

Seventh Edition

SAMS

800 East 96th Street, Indianapolis, Indiana 46240

Sams Teach Yourself C++ in One Hour a Day, Seventh Edition

Copyright © 2012 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-33567-9

ISBN-10: 0-672-33567-0

The Library of Congress Cataloging-in-Publication Data is on file.

Printed in the United States of America

Third Printing April 2013

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

international@pearsoned.com

Acquisitions Editor

Mark Taber

Development Editor

Songlin Qiu

Managing Editor

Sandra Schroeder

Project Editor

Mandie Frank

Copy Editor

Charlotte Kughen

Indexer

Tim Wright

Proofreader

Megan Wade

Technical Editor

Jon Upchurch

Publishing

Coordinator

Vanessa Evans

Designer

Gary Adair

Compositor

Studio Galou, LLC

Contents at a Glance

Introduction	1
PART I: The Basics	
1 Getting Started	5
2 The Anatomy of a C++ Program	15
3 Using Variables, Declaring Constants	29
4 Managing Arrays and Strings	57
5 Working with Expressions, Statements, and Operators	77
6 Controlling Program Flow	105
7 Organizing Code with Functions	141
8 Pointers and References Explained	165
PART II: Fundamentals of Object-Oriented C++ Programming	
9 Classes and Objects	203
10 Implementing Inheritance	251
11 Polymorphism	283
12 Operator Types and Operator Overloading	311
13 Casting Operators	353
14 An Introduction to Macros and Templates	367
PART III: Learning the Standard Template Library (STL)	
15 An Introduction to the Standard Template Library	393
16 The STL String Class	405
17 STL Dynamic Array Classes	423
18 STL list and forward_list	445
19 STL Set Classes	467
20 STL Map Classes	487
PART IV: More STL	
21 Understanding Function Objects	511
22 C++11 Lambda Expressions	527
23 STL Algorithms	543
24 Adaptive Containers: Stack and Queue	579
25 Working with Bit Flags Using STL	597
PART V: Advanced C++ Concepts	
26 Understanding Smart Pointers	607
27 Using Streams for Input and Output	621

28 Exception Handling	643
29 Going Forward	659

Appendixes

A Working with Numbers: Binary and Hexadecimal	671
B C++ Keywords	677
C Operator Precedence	679
D Answers	681
E ASCII Codes	723
Index	727

Table of Contents

Introduction	1
PART I: The Basics	
LESSON 1: Getting Started	5
A Brief History of C++	6
Connection to C	6
Advantages of C++	6
Evolution of the C++ Standard	7
Who Uses Programs Written in C++?	7
Programming a C++ Application	7
Steps to Generating an Executable	8
Analyzing Errors and Firefighting	8
Integrated Development Environments	8
Programming Your First C++ Application	9
Building and Executing Your First C++ Application	10
Understanding Compiler Errors	12
What's New in C++11	12
Summary	13
Q&A	13
Workshop	14
LESSON 2: The Anatomy of a C++ Program	15
Part of the Hello World Program	16
Preprocessor Directive #include	16
The Body of Your Program main()	17
Returning a Value	18
The Concept of Namespaces	19
Comments in C++ Code	20
Functions in C++	21
Basic Input Using std::cin and Output Using std::cout	24
Summary	26
Q&A	26
Workshop	27

LESSON 3: Using Variables, Declaring Constants	29
What Is a Variable?	30
Memory and Addressing in Brief	30
Declaring Variables to Access and Use Memory	30
Declaring and Initializing Multiple Variables of a Type	32
Understanding the Scope of a Variable	33
Global Variables	35
Common Compiler-Supported C++ Variable Types	36
Using Type bool to Store Boolean Values	37
Using Type char to Store Character Values	37
The Concept of Signed and Unsigned Integers	38
Signed Integer Types short, int, long, and long long	39
Unsigned Integer Types unsigned short, unsigned int, unsigned long, and unsigned long long	39
Floating-Point Types float and double	40
Determining the Size of a Variable Using sizeof	40
Using typedef to Substitute a Variable's Type	44
What Is a Constant?	45
Literal Constants	45
Declaring Variables as Constants Using const	46
Declaring Constants Using constexpr	47
Enumerated Constants	48
Defining Constants Using #define	50
Naming Variables and Constants	51
Keywords You Cannot Use as Variable or Constant Names	52
Summary	53
Q&A	53
Workshop	55
LESSON 4: Managing Arrays and Strings	57
What Is an Array?	58
The Need for Arrays	58
Declaring and Initializing Static Arrays	59
How Data Is Stored in an Array	60
Accessing Data Stored in an Array	61
Modifying Data Stored in an Array	62

Multidimensional Arrays	65
Declaring and Initializing Multidimensional Arrays	65
Accessing Elements in a Multidimensional Array	66
Dynamic Arrays	68
C-style Strings	70
C++ Strings: Using std::string	72
Summary	75
Q&A	75
Workshop	76
LESSON 5: Working with Expressions, Statements, and Operators	77
Statements	78
Compound Statements or Blocks	79
Using Operators	79
The Assignment Operator (=)	79
Understanding l-values and r-values	79
Operators to Add (+), Subtract (-), Multiply (*), Divide (/), and Modulo Divide (%)	80
Operators to Increment (++) and Decrement (--)	81
To Postfix or to Prefix?	81
Equality Operators (==) and (!=)	84
Relational Operators	85
Logical Operations NOT, AND, OR, and XOR	87
Using C++ Logical Operators NOT (!), AND (&&), and OR ()	88
Bitwise NOT (~), AND (&), OR (), and XOR (^) Operators	92
Bitwise Right Shift (>>) and Left Shift (<<) Operators	94
Compound Assignment Operators	96
Using Operator sizeof to Determine the Memory Occupied by a Variable	98
Operator Precedence	99
Summary	101
Q&A	102
Workshop	102
LESSON 6: Controlling Program Flow	105
Conditional Execution Using if ... else	106
Conditional Programming Using if ... else	107
Executing Multiple Statements Conditionally	109

Nested if Statements	111
Conditional Processing Using switch-case	115
Conditional Execution Using Operator (?:)	118
Getting Code to Execute in Loops	119
A Rudimentary Loop Using goto	119
The while Loop	121
The do...while loop	123
The for Loop	125
Modifying Loop Behavior Using continue and break	128
Loops That Don't End, that is, Infinite Loops	129
Controlling Infinite Loops	130
Programming Nested Loops	133
Using Nested Loops to Walk a Multidimensional Array	134
Using Nested Loops to Calculate Fibonacci Numbers	136
Summary	137
Q&A	138
Workshop	138

LESSON 7: Organizing Code with Functions **141**

The Need for Functions	142
What Is a Function Prototype?	143
What Is a Function Definition?	144
What Is a Function Call, and What Are Arguments?	144
Programming a Function with Multiple Parameters	145
Programming Functions with No Parameters or No Return Values	146
Function Parameters with Default Values	147
Recursion—Functions That Invoke Themselves	149
Functions with Multiple Return Statements	151
Using Functions to Work with Different Forms of Data	152
Overloading Functions	152
Passing an Array of Values to a Function	154
Passing Arguments by Reference	156
How Function Calls Are Handled by the Microprocessor	158
Inline Functions	159
Lambda Functions	161
Summary	162

Q&A	163
Workshop	163
LESSON 8: Pointers and References Explained	165
What Is a Pointer?	166
Declaring a Pointer	166
Determining the Address of a Variable Using the Reference Operator (&)	167
Using Pointers to Store Addresses	168
Access Pointed Data Using the Dereference Operator (*)	170
What Is the sizeof() of a Pointer?	173
Dynamic Memory Allocation	175
Using Operators new and delete to Allocate and Release Memory Dynamically	175
Effect of Incrementing and Decrementing Operators (++ and --) on Pointers	179
Using const Keyword on Pointers	181
Passing Pointers to Functions	182
Similarities Between Arrays and Pointers	184
Common Programming Mistakes When Using Pointers	186
Memory Leaks	187
When Pointers Don't Point to Valid Memory Locations	187
Dangling Pointers (Also Called Stray or Wild Pointers)	189
Pointer Programming Best-Practices	189
Checking If Allocation Request Using new Succeeded	191
What Is a Reference?	193
What Makes References Useful?	194
Using Keyword const on References	196
Passing Arguments by Reference to Functions	196
Summary	198
Q&A	198
Workshop	200
PART II: Fundamentals of Object-Oriented C++ Programming	
LESSON 9: Classes and Objects	203
The Concept of Classes and Objects	204
Declaring a Class	204
Instantiating an Object of a Class	205

Accessing Members Using the Dot Operator	206
Accessing Members Using the Pointer Operator (->)	206
Keywords public and private	208
Abstraction of Data via Keyword private	210
Constructors	212
Declaring and Implementing a Constructor	212
When and How to Use Constructors	213
Overloading Constructors	215
Class Without a Default Constructor	217
Constructor Parameters with Default Values	219
Constructors with Initialization Lists	220
Destructor	222
Declaring and Implementing a Destructor	222
When and How to Use Destructors	223
Copy Constructor	225
Shallow Copying and Associated Problems	225
Ensuring Deep Copy Using a Copy Constructor	228
Move Constructors Help Improve Performance	233
Different Uses of Constructors and Destructor	235
Class That Does Not Permit Copying	235
Singleton Class That Permits a Single Instance	236
Class That Prohibits Instantiation on the Stack	239
this Pointer	241
sizeof() a Class	242
How struct Differs from class	244
Declaring a friend of a class	245
Summary	247
Q&A	248
Workshop	249
LESSON 10: Implementing Inheritance	251
Basics of Inheritance	252
Inheritance and Derivation	252
C++ Syntax of Derivation	254
Access Specifier Keyword protected	256
Base Class Initialization—Passing Parameters to the Base Class	258

Derived Class Overriding Base Class' Methods	261
Invoking Overridden Methods of a Base Class	263
Invoking Methods of a Base Class in a Derived Class	264
Derived Class Hiding Base Class' Methods	266
Order of Construction	268
Order of Destruction	268
Private Inheritance	271
Protected Inheritance	273
The Problem of Slicing	277
Multiple Inheritance	277
Summary	281
Q&A	281
Workshop	281
LESSON 11: Polymorphism	283
Basics of Polymorphism	284
Need for Polymorphic Behavior	284
Polymorphic Behavior Implemented Using Virtual Functions	286
Need for Virtual Destructors	288
How Do virtual Functions Work? Understanding the Virtual Function Table	292
Abstract Base Classes and Pure Virtual Functions	296
Using virtual Inheritance to Solve the Diamond Problem	299
Virtual Copy Constructors?	304
Summary	307
Q&A	307
Workshop	308
LESSON 12: Operator Types and Operator Overloading	311
What Are Operators in C++?	312
Unary Operators	313
Types of Unary Operators	313
Programming a Unary Increment/Decrement Operator	314
Programming Conversion Operators	317
Programming Dereference Operator (*) and Member Selection	
Operator (->)	319
Binary Operators	323
Types of Binary Operators	324

Programming Binary Addition (a+b) and Subtraction (a-b) Operators	325
Implementing Addition Assignment (+=) and Subtraction Assignment (-=) Operators	327
Overloading Equality (==) and Inequality (!=) Operators	330
Overloading <, >, <=, and >= Operators	332
Overloading Copy Assignment Operator (=)	335
Subscript Operator ([])	338
Function Operator ()	342
Operators That Cannot Be Overloaded	349
Summary	350
Q&A	351
Workshop	351
LESSON 13: Casting Operators	353
The Need for Casting	354
Why C-Style Casts Are Not Popular with Some C++ Programmers	355
The C++ Casting Operators	355
Using static_cast	356
Using dynamic_cast and Runtime Type Identification	357
Using reinterpret_cast	360
Using const_cast	361
Problems with the C++ Casting Operators	362
Summary	363
Q&A	364
Workshop	364
LESSON 14: An Introduction to Macros and Templates	367
The Preprocessor and the Compiler	368
Using #define Macros to Define Constants	368
Using Macros for Protection Against Multiple Inclusion	371
Using #define To Write Macro Functions	372
Why All the Parentheses?	374
Using Macro assert to Validate Expressions	375
Advantages and Disadvantages of Using Macro Functions	376
An Introduction to Templates	378
Template Declaration Syntax	378
The Different Types of Template Declarations	379
Template Functions	379

Templates and Type Safety	381
Template Classes	381
Template Instantiation and Specialization	383
Declaring Templates with Multiple Parameters	383
Declaring Templates with Default Parameters	384
Sample Template class<> HoldsPair	385
Template Classes and static Members	386
Using Templates in Practical C++ Programming	389
Summary	390
Q&A	390
Workshop	391

PART III: Learning the Standard Template Library (STL)

LESSON 15: An Introduction to the Standard Template Library 393

STL Containers	394
Sequential Containers	394
Associative Containers	395
Choosing the Right Container	396
STL Iterators	399
STL Algorithms	400
The Interaction Between Containers and Algorithms Using Iterators	400
STL String Classes	403
Summary	403
Q&A	403
Workshop	404

LESSON 16: The STL String Class 405

The Need for String Manipulation Classes	406
Working with the STL String Class	407
Instantiating the STL String and Making Copies	407
Accessing Character Contents of a std::string	410
Concatenating One String to Another	412
Finding a Character or Substring in a String	413
Truncating an STL string	415
String Reversal	417
String Case Conversion	418

Template-Based Implementation of an STL String	420
Summary	420
Q&A	421
Workshop	421
LESSON 17: STL Dynamic Array Classes	423
The Characteristics of <code>std::vector</code>	424
Typical Vector Operations	424
Instantiating a Vector	424
Inserting Elements at the End Using <code>push_back()</code>	426
Inserting Elements at a Given Position Using <code>insert()</code>	428
Accessing Elements in a Vector Using Array Semantics	431
Accessing Elements in a Vector Using Pointer Semantics	433
Removing Elements from a Vector	434
Understanding the Concepts of Size and Capacity	436
The STL <code>deque</code> Class	438
Summary	441
Q&A	441
Workshop	442
LESSON 18: STL <code>list</code> and <code>forward_list</code>	445
The Characteristics of a <code>std::list</code>	446
Basic <code>list</code> Operations	446
Instantiating a <code>std::list</code> Object	446
Inserting Elements at the Front or Back of the List	448
Inserting at the Middle of the List	450
Erasing Elements from the List	453
Reversing and Sorting Elements in a List	455
Reversing Elements Using <code>list::reverse()</code>	455
Sorting Elements	456
Sorting and Removing Elements from a list That Contains Objects of a class	458
Summary	465
Q&A	465
Workshop	465

LESSON 19: STL Set Classes	467
An Introduction to STL Set Classes	468
Basic STL set and multiset Operations	468
Instantiating a <code>std::set</code> Object	469
Inserting Elements in a set or multiset	471
Finding Elements in an STL set or multiset	473
Erasing Elements in an STL set or multiset	475
Pros and Cons of Using STL set and multiset	480
Summary	484
Q&A	484
Workshop	485
LESSON 20: STL Map Classes	487
An Introduction to STL Map Classes	488
Basic <code>std::map</code> and <code>std::multimap</code> Operations	489
Instantiating a <code>std::map</code> or <code>std::multimap</code>	489
Inserting Elements in an STL map or multimap	491
Finding Elements in an STL map	494
Finding Elements in an STL multimap	496
Erasing Elements from an STL map or multimap	497
Supplying a Custom Sort Predicate	499
How Hash Tables Work	504
Using C++11 Hash Tables: <code>unordered_map</code> and <code>unordered_multimap</code>	504
Summary	508
Q&A	509
Workshop	510
PART IV: More STL	
LESSON 21: Understanding Function Objects	511
The Concept of Function Objects and Predicates	512
Typical Applications of Function Objects	512
Unary Functions	512
Unary Predicate	517
Binary Functions	519
Binary Predicate	522

Summary	524
Q&A	524
Workshop	525
LESSON 22: C++11 Lambda Expressions	527
What Is a Lambda Expression?	528
How to Define a Lambda Expression	529
Lambda Expression for a Unary Function	529
Lambda Expression for a Unary Predicate	531
Lambda Expression with State via Capture Lists [...]	532
The Generic Syntax of Lambda Expressions	534
Lambda Expression for a Binary Function	535
Lambda Expression for a Binary Predicate	537
Summary	540
Q&A	541
Workshop	541
LESSON 23: STL Algorithms	543
What Are STL Algorithms?	544
Classification of STL Algorithms	544
Non-Mutating Algorithms	544
Mutating Algorithms	545
Usage of STL Algorithms	547
Finding Elements Given a Value or a Condition	547
Counting Elements Given a Value or a Condition	550
Searching for an Element or a Range in a Collection	552
Initializing Elements in a Container to a Specific Value	554
Using <code>std::generate()</code> to Initialize Elements to a Value Generated at Runtime	556
Processing Elements in a Range Using <code>for_each()</code>	557
Performing Transformations on a Range Using <code>std::transform()</code>	560
Copy and Remove Operations	562
Replacing Values and Replacing Element Given a Condition	565
Sorting and Searching in a Sorted Collection and Erasing Duplicates	567
Partitioning a Range	570
Inserting Elements in a Sorted Collection	572
Summary	575
Q&A	575
Workshop	576

LESSON 24: Adaptive Containers: Stack and Queue	579
The Behavioral Characteristics of Stacks and Queues	580
Stacks	580
Queues	580
Using the STL stack Class	581
Instantiating the Stack	581
Stack Member Functions	582
Insertion and Removal at Top Using push() and pop()	583
Using the STL queue Class	585
Instantiating the Queue	585
Member Functions of a queue	586
Insertion at End and Removal at the Beginning of queue via push() and pop()	587
Using the STL Priority Queue	589
Instantiating the priority_queue Class	589
Member Functions of priority_queue	590
Insertion at the End and Removal at the Beginning of priority_queue via push() and pop()	591
Summary	594
Q&A	594
Workshop	594
LESSON 25: Working with Bit Flags Using STL	597
The bitset Class	598
Instantiating the std::bitset	598
Using std::bitset and Its Members	599
Useful Operators Featured in std::bitset	599
std::bitset Member Methods	600
The vector<bool>	603
Instantiating vector<bool>	603
vector<bool> Functions and Operators	604
Summary	605
Q&A	605
Workshop	606

PART V: Advanced C++ Concepts

LESSON 26: Understanding Smart Pointers	607
What Are Smart Pointers?	608
The Problem with Using Conventional (Raw) Pointers	608
How Do Smart Pointers Help?	608
How Are Smart Pointers Implemented?	609
Types of Smart Pointers	610
Deep Copy	611
Copy on Write Mechanism	613
Reference-Counted Smart Pointers	613
Reference-Linked Smart Pointers	614
Destructive Copy	614
Using the <code>std::unique_ptr</code>	617
Popular Smart Pointer Libraries	618
Summary	619
Q&A	619
Workshop	620
LESSON 27: Using Streams for Input and Output	621
Concept of Streams	622
Important C++ Stream Classes and Objects	623
Using <code>std::cout</code> for Writing Formatted Data to Console	624
Changing Display Number Formats Using <code>std::cout</code>	624
Aligning Text and Setting Field Width Using <code>std::cout</code>	627
Using <code>std::cin</code> for Input	628
Using <code>std::cin</code> for Input into a Plain Old Data Type	628
Using <code>std::cin::get</code> for Input into C-Style char Buffer	629
Using <code>std::cin</code> for Input into a <code>std::string</code>	630
Using <code>std::fstream</code> for File Handling	632
Opening and Closing a File Using <code>open()</code> and <code>close()</code>	632
Creating and Writing a Text File Using <code>open()</code> and <code>operator<<</code>	634
Reading a Text File Using <code>open()</code> and <code>operator>></code>	635
Writing to and Reading from a Binary File	636
Using <code>std::stringstream</code> for String Conversions	638
Summary	640
Q&A	640
Workshop	641

LESSON 28: Exception Handling	643
What Is an Exception?	644
What Causes Exceptions?	644
Implementing Exception Safety via try and catch	645
Using catch(...) to Handle All Exceptions	645
Catching Exception of a Type	647
Throwing Exception of a Type Using throw	648
How Exception Handling Works	650
Class std::exception	652
Your Custom Exception Class Derived from std::exception	653
Summary	655
Q&A	656
Workshop	656
LESSON 29: Going Forward	659
What's Different in Today's Processors?	660
How to Better Use Multiple Cores	661
What Is a Thread?	661
Why Program Multithreaded Applications?	662
How Can Threads Transact Data?	663
Using Mutexes and Semaphores to Synchronize Threads	664
Problems Caused by Multithreading	664
Writing Great C++ Code	665
Learning C++ Doesn't Stop Here!	667
Online Documentation	667
Communities for Guidance and Help	668
Summary	668
Q&A	668
Workshop	669
Appendixes	
APPENDIX A: Working with Numbers: Binary and Hexadecimal	671
Decimal Numeral System	672
Binary Numeral System	672
Why Do Computers Use Binary?	673

What Are Bits and Bytes?	673
How Many Bytes Make a Kilobyte?.....	674
Hexadecimal Numeral System	674
Why Do We Need Hexadecimal?	674
Converting to a Different Base	675
The Generic Conversion Process	675
Converting Decimal to Binary	675
Converting Decimal to Hexadecimal	676
APPENDIX B: C++ Keywords	677
APPENDIX C: Operator Precedence	679
APPENDIX D: Answers	681
APPENDIX E: ASCII Codes	723
ASCII Table of Printable Characters	724
Index	727

About the Author

Siddhartha Rao is a technologist at SAP AG, the world's leading supplier of enterprise software. As the head of SAP Product Security India, his primary responsibilities include hiring expert talent in the area of product security as well as defining development best practices that keeps SAP software globally competitive. Awarded Most Valuable Professional by Microsoft for Visual Studio–Visual C++, he is convinced that C++11 will help you program faster, simpler, and more efficient C++ applications.

Siddhartha also loves traveling and discovering new cultures given an opportunity to. For instance, parts of this book have been composed facing the Atlantic Ocean at a quaint village called Plogoff in Brittany, France—one of the four countries this book was authored in. He looks forward to your feedback on this global effort!

Dedication

This book is dedicated to my lovely parents and my wonderful sister for being there when I have needed them the most.

Acknowledgments

I am deeply indebted to my friends who cooked and baked for me while I burned the midnight oil working on this project. I am grateful to the editorial staff for their very professional engagement and the wonderful job in getting this book to your shelf!

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write directly to let us know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone number or email address.

E-mail: feedback@sampublishing.com

Mail: Reader Feedback
 Sams Publishing
 800 East 96th Street
 Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

2011 was a special year for C++. With the ratification of the new standard, C++11 empowers you to write better code using new keywords and constructs that increase your programming efficiency. This book helps you learn C++11 in tiny steps. It has been thoughtfully divided into lessons that teach you the fundamentals of this object-oriented programming language from a practical point of view. Depending on your proficiency level, you will be able to master C++11 one hour at a time.

Learning C++ by doing is the best way—so try the rich variety of code samples in this book hands-on and help yourself improve your programming proficiency. These code snippets have been tested using the latest versions of the available compilers at the time of writing, namely the Microsoft Visual C++ 2010 compiler for C++ and GNU's C++ compiler version 4.6, which both offer a rich coverage of C++11 features.

Who Should Read This Book?

The book starts with the very basics of C++. All that is needed is a desire to learn this language and curiosity to understand how stuff works. An existing knowledge of C++ programming can be an advantage but is not a prerequisite. This is also a book you might like to refer to if you already know C++ but want to learn additions that have been made to the language in C++11. If you are a professional programmer, Part III, “Learning the Standard Template Library (STL),” is bound to help you create better, more practical C++11 applications.

Organization of This Book

Depending on your current proficiency levels with C++, you can choose the section you would like to start with. This book has been organized into five parts:

- Part I, “The Basics,” gets you started with writing simple C++ applications. In doing so, it introduces you to the keywords that you most frequently see in C++ code of a variable without compromising on type safety.
- Part II, “Fundamentals of Object-Oriented C++ Programming,” teaches you the concept of classes. You learn how C++ supports the important object-oriented programming principles of encapsulation, abstraction, inheritance, and polymorphism.

Lesson 9, “Classes and Objects,” teaches you the new C++11 concept of move constructor followed by the move assignment operator in Lesson 12, “Operator Types and Operator Overloading.” These performance features help reduce unwanted and unnecessary copy steps, boosting the performance of your application. Lesson 14, “An Introduction to Macros and Templates,” is your stepping stone into writing powerful generic C++ code.

- Part III, “Learning the Standard Template Library (STL),” helps you write efficient and practical C++ code using the STL `string` class and containers. You learn how `std::string` makes simple string concatenation operations safe and easy and how you don’t need to use C-style `char*` strings anymore. You will be able to use STL dynamic arrays and linked lists instead of programming your own.
- Part IV, “More STL,” focuses on algorithms. You learn to use `sort` on containers such as `vector` via iterators. In this part, you find out how C++11 keyword `auto` has made a significant reduction to the length of your iterator declarations. Lesson 22, “C++11 Lambda Expressions,” presents a powerful new feature that results in significant code reduction when you use STL algorithms.
- Part V, “Advanced C++ Concepts,” explains language capabilities such as smart pointers and exception-handling, which are not a must in a C++ application but help make a significant contribution toward increasing its stability and quality. This part ends with a note on best practices in writing good C++11 applications.

Conventions Used in This Book

Within the lessons, you find the following elements that provide additional information:

NOTE

These boxes provide additional information related to material you read.

C++11

These boxes highlight features new to C++11. You may need to use the newer versions of the available compilers to use these language capabilities.

CAUTION

These boxes alert your attention to problems or side effects that can occur in special situations.

TIP

These boxes give you best practices in writing your C++ programs.

DO

DO use the “Do/Don’t” boxes to find a quick summary of a fundamental principle in a lesson.

DON'T

DON'T overlook the useful information offered in these boxes.

This book uses different typefaces to differentiate between code and plain English. Throughout the lessons, code, commands, and programming-related terms appear in a computer typeface.

Sample Code for this Book

The code samples in this book are available online for download from the publisher’s website.

This page intentionally left blank

LESSON 2

The Anatomy of a C++ Program

C++ programs consist of classes, functions, variables, and other component parts. Most of this book is devoted to explaining these parts in depth, but to get a sense of how a program fits together, you must see a complete working program.

In this lesson, you learn

- The parts of a C++ program
- How the parts work together
- What a function is and what it does
- Basic input and output operations

Part of the Hello World Program

Your first C++ program in Lesson 1, “Getting Started,” did nothing more than write a simple “Hello World” statement to the screen. Yet this program contains some of the most important and basic building blocks of a C++ program. You use Listing 2.1 as a starting point to analyze components all C++ programs contain.

LISTING 2.1 HelloWorldAnalysis.cpp: Analyze a C++ Program

```
1: // Preprocessor directive that includes header iostream
2: #include <iostream>
3:
4: // Start of your program: function block main()
5: int main()
6: {
7:     /* Write to the screen */
8:     std::cout << "Hello World" << std::endl;
9:
10:    // Return a value to the OS
11:    return 0;
12: }
```

This C++ program can be broadly classified into two parts: the preprocessor directives that start with a # and the main body of the program that starts with `int main()`.

NOTE

Lines 1, 4, 7, and 10, which start with a // or with a /*, are comments and are ignored by the compiler. These comments are for humans to read.

Comments are discussed in greater detail in the next section.

Preprocessor Directive #include

As the name suggests, a *preprocessor* is a tool that runs before the actual compilation starts. Preprocessor directives are commands to the preprocessor and always start with a pound sign #. In Line 2 of Listing 2.1, `#include <filename>` tells the preprocessor to take the contents of the file (`iostream`, in this case) and include them at the line where the directive is made. `iostream` is a standard header file that is included because it contains the definition of `std::cout` used in Line 8 that prints “Hello World” on the screen. In other words, the compiler was able to compile Line 8 that contains `std::cout` because we instructed the preprocessor to include the definition of `std::cout` in Line 2.

NOTE

In professionally programmed C++ applications, not all includes are only standard headers. Complex applications are typically programmed in multiple files wherein some need to include others. So, if an artifact declared in FileA needs to be used in FileB, you need to include the former in the latter. You usually do that by putting the following `include` statement in FileA:

```
#include "...relative path to FileB\FileB"
```

We use quotes in this case and not angle brackets in including a self-created header. `<>` brackets are typically used when including standard headers.

2

The Body of Your Program `main()`

Following the preprocessor directive(s) is the body of the program characterized by the function `main()`. The execution of a C++ program always starts here. It is a standardized convention that function `main()` is declared with an `int` preceding it. `int` is the return value type of the function `main()`.

NOTE

In many C++ applications, you find a variant of the `main()` function that looks like this:

```
int main (int argc, char* argv[])
```

This is also standard compliant and acceptable as `main` returns `int`. The contents of the parenthesis are “arguments” supplied to the program. This program possibly allows the user to start it with command-line arguments, such as

```
program.exe /DoSomethingSpecific
```

`/DoSomethingSpecific` is the argument for that program passed by the OS as a parameter to it, to be handled within `main`.

Let’s discuss Line 8 that fulfills the actual purpose of this program!

```
std::cout << "Hello World" << std::endl;
```

`cout` (“console-out”, also pronounced see-out) is the statement that writes “Hello World” to the screen. `cout` is a stream defined in the standard namespace (hence, `std::cout`), and what you are doing in this line is putting the text “Hello World” into this stream by using the stream insertion operator `<<`. `std::endl` is used to end a line, and inserting it into a stream is akin to inserting a carriage return. Note that the stream insertion operator is used every time a new entity needs to be inserted into the stream.

The good thing about streams in C++ is that similar stream semantics used with another stream type result in a different operation being performed with the same text—for example, insertion into a file instead of a console. Thus, working with streams gets intuitive, and when you are used to one stream (such as `cout` that writes text to the console), you find it easy to work with others (such as `fstream` that helps write text files to the disk).

Streams are discussed in greater detail in Lesson 27, “Using Streams for Input and Output.”

NOTE

The actual text, including the quotes "Hello World", is called a *string literal*.

Returning a Value

Functions in C++ need to return a value unless explicitly specified otherwise. `main()` is a function, too, and always returns an integer. This value is returned to the operating system (OS) and, depending on the nature of your application, can be very useful as most OSes provide for an ability to query on the return value of an application that has terminated naturally. In many cases, one application is launched by another and the parent application (that launches) wants to know if the child application (that was launched) has completed its task successfully. The programmer can use the return value of `main()` to convey a success or error state to the parent application.

NOTE

Conventionally programmers return 0 in the event of success or -1 in the event of error. However, the return value is an integer, and the programmer has the flexibility to convey many different states of success or failure using the available range of integer return values.

CAUTION

C++ is case-sensitive. So, expect compilation to fail if you write `Int` instead of `int`, `Void` instead of `void`, and `Std::Cout` instead of `std::cout`.

The Concept of Namespaces

The reason you used `std::cout` in the program and not only `cout` is that the artifact (`cout`) that you want to invoke is in the standard (`std`) namespace.

So, what exactly are namespaces?

Assume that you didn't use the namespace qualifier in invoking `cout` and assume that `cout` existed in two locations known to the compiler—which one should the compiler invoke? This causes a conflict and the compilation fails, of course. This is where namespaces get useful. Namespaces are names given to parts of code that help in reducing the potential for a naming conflict. By invoking `std::cout`, you are telling the compiler to use that one unique `cout` that is available in the `std` namespace.

NOTE

You use the `std` (pronounced “standard”) namespace to invoke functions, streams, and utilities that have been ratified by the ISO Standards Committee and are hence declared within it.

Many programmers find it tedious to repeatedly add the `std` namespace specifier to their code when using `cout` and other such features contained in the same. The `using` namespace declaration as demonstrated in Listing 2.2 will help you avoid this repetition.

LISTING 2.2 The `using` namespace Declaration

```
1: // Pre-processor directive
2: #include <iostream>
3:
4: // Start of your program
5: int main()
6: {
7:     // Tell the compiler what namespace to search in
8:     using namespace std;
9:
10:    /* Write to the screen using std::cout */
11:    cout << "Hello World" << endl;
12:
13:    // Return a value to the OS
14:    return 0;
15: }
```


Analysis ▼

Note Line 8. By telling the compiler that you are using the namespace `std`, you don't need to explicitly mention the namespace on Line 11 when using `std::cout` or `std::endl`.

A more restrictive variant of Listing 2.2 is shown in Listing 2.3 where you do not include a namespace in its entirety. You only include those artifacts that you wish to use.

LISTING 2.3 Another Demonstration of the `using` Keyword

```
1: // Pre-processor directive
2: #include <iostream>
3:
4: // Start of your program
5: int main()
6: {
7:     using std::cout;
8:     using std::endl;
9:
10:    /* Write to the screen using cout */
11:    cout << "Hello World" << endl;
12:
13:    // Return a value to the OS
14:    return 0;
15: }
```

Analysis ▼

Line 8 in Listing 2.2 has now been replaced by Lines 7 and 8 in Listing 2.3. The difference between using namespace `std` and using `std::cout` is that the former allows all artifacts in the `std` namespace to be used without explicitly needing to specify the namespace qualifier `std::`. With the latter, the convenience of not needing to disambiguate the namespace explicitly is restricted to only `std::cout` and `std::endl`.

Comments in C++ Code

Lines 1, 4, 10 and 13 in Listing 2.3 contain text in a spoken language (English, in this case) yet do not interfere with the ability of the program to compile. They also do not alter the output of the program. Such lines are called *comments*. Comments are ignored by the compiler and are popularly used by programmers to explain their code—hence, they are written in human- (or geek-) readable language.

C++ supports comments in two styles:

- `//` indicates that the line is a comment. For example:

```
// This is a comment
```
- `/*` followed by `*/` indicates the contained text is a comment, even if it spans multiple lines:

```
/* This is a comment
and it spans two lines */
```

NOTE

It might seem strange that a programmer needs to explain his own code, but the bigger a program gets or the larger the number of programmers working on a particular module gets, the more important it is to write code that can be easily understood. It is important to explain what is being done and why it is being done in that particular manner using well-written comments.

DO	DON'T
<p>Do add comments explaining the working of complicated algorithms and complex parts of your program.</p> <p>Do compose comments in a style that fellow programmers can understand.</p>	<p>Don't use comments to explain or repeat the obvious.</p> <p>Don't forget that adding comments will not justify writing obscure code.</p> <p>Don't forget that when code is modified, comments might need to be updated, too.</p>

Functions in C++

Functions in C++ are the same as functions in C. Functions are artifacts that enable you to divide the content of your application into functional units that can be invoked in a sequence of your choosing. A function, when called (that is, invoked), typically returns a value to the calling function. The most famous function is, of course, `main()`. It is recognized by the compiler as the starting point of your C++ application and has to return an `int` (i.e., an integer).

You as a programmer have the choice and usually the need to compose your own functions. Listing 2.4 is a simple application that uses a function to display statements on the screen using `std::cout` with various parameters.

LISTING 2.4 Declaring, Defining, and Calling a Function That Demonstrates Some Capabilities of `std::cout`

```
1: #include <iostream>
2: using namespace std;
3:
4: // Function declaration
5: int DemoConsoleOutput();
6:
7: int main()
8: {
9:     // Call i.e. invoke the function
10:    DemoConsoleOutput();
11:
12:    return 0;
13: }
14:
15: // Function definition
16: int DemoConsoleOutput()
17: {
18:    cout << "This is a simple string literal" << endl;
19:    cout << "Writing number five: " << 5 << endl;
20:    cout << "Performing division 10 / 5 = " << 10 / 5 << endl;
21:    cout << "Pi when approximated is 22 / 7 = " << 22 / 7 << endl;
22:    cout << "Pi more accurately is 22 / 7 = " << 22.0 / 7 << endl;
23:
24:    return 0;
25: }
```

Output ▶

```
This is a simple string literal
Writing number five: 5
Performing division 10 / 5 = 2
Pi when approximated is 22 / 7 = 3
Pi more accurately is 22 / 7 = 3.14286
```

Analysis ▶

Lines 5, 10, and 15 through 25 are those of interest. Line 5 is called a *function declaration*, which basically tells the compiler that you want to create a function called `DemoConsoleOutput()` that returns an `int` (integer). It is because of this *declaration* that the compiler agrees to compile Line 10, assuming that the *definition* (that is, the implementation of the function) comes up, which it does in Lines 15 through 25.

This function actually displays the various capabilities of `cout`. Note how it not only prints text the same way as it displayed “Hello World” in previous examples, but also the

result of simple arithmetic computations. Lines 21 and 22 both attempt to display the result of pi ($22 / 7$), but the latter is more accurate simply because by dividing 22.0 by 7, you tell the compiler to treat the result as a real number (a `float` in C++ terms) and not as an integer.

Note that your function is stipulated to return an integer and returns 0. As it did not perform any decision-making, there was no need to return any other value. Similarly, `main()` returns 0, too. Given that `main()` has delegated all its activity to the function `DemoConsoleOutput()`, you would be wiser to use the return value of the function in returning from `main()` as seen in Listing 2.5.

2

LISTING 2.5 Using the Return Value of a Function

```
1: #include <iostream>
2: using namespace std;
3:
4: // Function declaration and definition
5: int DemoConsoleOutput()
6: {
7:     cout << "This is a simple string literal" << endl;
8:     cout << "Writing number five: " << 5 << endl;
9:     cout << "Performing division 10 / 5 = " << 10 / 5 << endl;
10:    cout << "Pi when approximated is 22 / 7 = " << 22 / 7 << endl;
11:    cout << "Pi more accurately is 22 / 7 = " << 22.0 / 7 << endl;
12:
13:    return 0;
14: }
15:
16: int main()
17: {
18:     // Function call with return used to exit
19:     return DemoConsoleOutput();
20: }
```

Analysis ►

The output of this application is the same as the output of the previous listing. Yet, there are slight changes in the way it is programmed. For one, as you have defined (i.e., implemented) the function before `main()` at Line 5, you don't need an extra declaration of the same. Modern C++ compilers take it as a function declaration and definition in one. `main()` is a bit shorter, too. Line 19 invokes the function `DemoConsoleOutput()` and simultaneously returns the return value of the function from the application.

NOTE

In cases such as this where a function is not required to make a decision or return success or failure status, you can declare a function of return type `void`:

```
void DemoConsoleOutput()
```

This function cannot return a value, and the execution of a function that returns `void` cannot be used to make a decision.

Functions can take parameters, can be recursive, can contain multiple return statements, can be overloaded, can be expanded in-line by the compiler, and lots more. These concepts are introduced in greater detail in Lesson 7, “Organizing Code with Functions.”

Basic Input Using `std::cin` and Output Using `std::cout`

Your computer enables you to interact with applications running on it in various forms and allows these applications to interact with you in many forms, too. You can interact with applications using the keyboard or the mouse. You can have information displayed on the screen as text, displayed in the form of complex graphics, printed on paper using a printer, or simply saved to the file system for later usage. This section discusses the very simplest form of input and output in C++—using the console to write and read information.

You use `std::cout` (pronounced “standard see-out”) to write simple text data to the console and use `std::cin` (“standard see-in”) to read text and numbers (entered using the keyboard) from the console. In fact, in displaying “Hello World” on the screen, you have already encountered `cout`, as seen in Listing 2.1:

```
8:    std::cout << "Hello World" << std::endl;
```

The statement shows `cout` followed by the insertion operator `<<` (that helps insert data into the output stream), followed by the string literal “Hello World” to be inserted, followed by a new line in the form of `std::endl` (pronounced “standard end-line”).

The usage of `cin` is simple, too, and as `cin` is used for input, it is accompanied by the variable you want to be storing the input data in:

```
std::cin >> Variable;
```

Thus, `cin` is followed by the extraction operator `>>` (extracts data from the input stream), which is followed by the variable where the data needs to be stored. If the user input

needs to be stored in two variables, each containing data separated by a space, then you can do so using one statement:

```
std::cin >> Variable1 >> Variable2;
```

Note that `cin` can be used for text as well as numeric inputs from the user, as shown in Listing 2.6.

LISTING 2.6 Use `cin` and `cout` to Display Number and Text Input by User

```
1: #include <iostream>
2: #include <string>
3: using namespace std;
4:
5: int main()
6: {
7:     // Declare a variable to store an integer
8:     int InputNumber;
9:
10:    cout << "Enter an integer: ";
11:
12:    // store integer given user input
13:    cin >> InputNumber;
14:
15:    // The same with text i.e. string data
16:    cout << "Enter your name: ";
17:    string InputName;
18:    cin >> InputName;
19:
20:    cout << InputName << " entered " << InputNumber << endl;
21:
22:    return 0;
23: }
```

Output ►

```
Enter an integer: 2011
Enter your name: Siddhartha
Siddhartha entered 2011
```

Analysis ►

Line 8 shows how a variable of name `InputNumber` is declared to store data of type `int`. The user is requested to enter a number using `cout` in Line 10, and the entered number is stored in the integer variable using `cin` in Line 13. The same exercise is repeated with storing the user's name, which of course cannot be held in an integer but in a different

type called `string` as seen in Lines 17 and 18. The reason you included `<string>` in Line 2 was to use type `string` later inside `main()`. Finally in Line 20, a `cout` statement is used to display the entered name with the number and an intermediate text to produce the output `Siddhartha entered 2011`.

This is a very simple example of how basic input and output work in C++. Don't worry if the concept of variables is not clear to you as it is explained in good detail in the following Lesson 3, "Using Variables, Declaring Constants."

Summary

This lesson introduced the basic parts of a simple C++ program. You understood what `main()` is, got an introduction to namespaces, and learned the basics of console input and output. You are able to use a lot of these in every program you write.

Q&A

Q What does `#include` do?

A This is a directive to the preprocessor that runs when you call your compiler. This specific directive causes the contents of the file named in `<>` after `#include` to be inserted at that line as if it were typed at that location in your source code.

Q What is the difference between `//` comments and `/*` comments?

A The double-slash comments (`//`) expire at the end of the line. Slash-star (`/*`) comments are in effect until there is a closing comment mark (`*/`). The double-slash comments are also referred to as *single-line comments*, and the slash-star comments are often referred to as *multiline comments*. Remember, not even the end of the function terminates a slash-star comment; you must put in the closing comment mark or you will receive a compile-time error.

Q When do you need to program command-line arguments?

A To allow the user to alter the behavior of a program. For example, the command `ls` in Linux or `dir` in Windows enables you to see the contents within the current directory or folder. To view files in another directory, you would specify the path of the same using command-line arguments, as seen in `ls /` or `dir \.`

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. Try to answer the quiz and exercise questions before checking the answers in Appendix D, and be certain you understand the answers before continuing to the next lesson.

Quiz

1. What is the problem in declaring `Int main()`?
2. Can comments be longer than one line?

2

Exercises

1. **BUG BUSTERS:** Enter this program and compile it. Why does it fail? How can you fix it?

```
1: #include <iostream>
2: void main()
3: {
4:     std::Cout << "Is there a bug here?";
5: }
```

2. Fix the bug in Exercise 1 and recompile, link, and run it.
3. Modify Listing 2.4 to demonstrate subtraction (using `-`) and multiplication (using `*`).

This page intentionally left blank

Index

Symbols

+= (addition assignment) operator, 327-329

{} (braces), executing multiple statements conditionally, 109-110

!= (equality) operator, 84

[] (subscript) operator, 338-341

-+ (subtraction assignment) operator, 327-329

+ (addition) operator, 80-81, 325-327

& (AND) operator, 92-94, 167-168

= (assignment) operator, 79

= (copy assignment) operator, overloading, 335-338

/ (divide) operator, 80-81

== (equality) operator, 84

% (modulo divide) operator, 80-81

***** (multiply operator), 80-81, 170-173

~ (NOT) operator, 92-94

| (OR) operator, 92-94

^ (XOR) operator, 92-94

? operator, 118-119

#define directive

- constants, defining, 50, 368-371
- macro functions, writing, 372-374

A

abstract base classes, 296-298

abstracting data via private keyword, 210-212

access specifiers, 256

accessing

- arrays, zero-based index, 61-62
- elements in vectors, 431-434
- memory with variables, 30-32
- multidimensional array elements, 66-68
- pointed data with dereference operator, 170-173
- STL string classes, 410-411

adaptive containers, 579. See also containers

- queues, 580-581
 - inserting/removing elements, 587-589
 - instantiating, 585-586
 - member functions, 587
 - priority queues, 589-594
- stacks, 580
 - inserting/removing elements, 583-585
 - instantiating, 581-582
 - member functions, 582

adaptive function objects, 512

addition (+) operator, 80-81, 325-327

addressing, 30, 61-62

advantages

- of C++, 6
- of macro functions, 377

algorithms

- containers, 547
- STL, 400, 543
 - classification of, 544-547
 - copy and remove operations, 562-565
 - counting and finding elements, 550
 - initializing elements, 554-557
 - inserting elements, 572-574
 - overview of, 544
 - partitioning ranges, 570-572
 - processing elements, 557-559
 - replacing elements, 565-567
 - searching ranges, 552
 - sorting collections, 567-570
 - transforming ranges, 560-562

aligning text, 627-628

allocating memory

- delete operator, 175-178
- new operator, 175-176

analyzing null terminator, 70-71

AND operator (&), 88-92

applications

- Hello World
 - compiling, 12
 - writing, 9-11
- multithreaded, 661
 - problems caused by, 664-665
 - programming, reasons for, 662-663

- writing, best practices, 665-666

applying

- const cast, 361-362
- dynamic cast, 357-360
- function objects
 - binary functions, 519-524
 - unary functions, 512-519
- reinterpret cast, 360
- static cast, 356
- STL string classes
 - accessing, 410-411
 - case conversion, 418-419
 - concatenation, 412
 - find member function, 413-415
 - instantiating, 407-409
 - reversing, 417-418
 - template-based implementation, 420
 - truncating, 415-417
- templates, 389

arguments, 144

- passing by reference, 156-157
- passing to functions, 196-198

arithmetic operators, 80-83

arrays, 47

- as function parameters, 154-155
- C-style strings, 70-72
- characters, declaring, 406
- data, accessing, 61-62
- dynamic, 68-69
- modifying data in, 62-65
- multidimensional
 - accessing elements in, 66-68
 - declaring, 65-66
 - iterating with nested loops, 134-135
- need for, 58-59
- organization in, 60

- pointers, similarity to, 184-186

static, declaring, 59-60

STL dynamic array class

- accessing elements in vectors, 431-434
- deleting elements from vectors, 434-435
- inserting elements into vectors, 426
- instantiating vectors, 424-425
- need for, 423
- size and capacity of vectors, 436-437
- vectors, 424

ASCII codes, 724-726

assert() macro, validating expressions, 376

assigning values to array elements, 62-65

assignment operators, 79, 96-98, 327-329

associative containers, 395

auto keyword, 42-44

B

base classes

- abstract base classes, 296-298
- the Diamond Problem, 303
- exceptions, 652-655
- initialization, 258-261
- overridden methods, invoking, 263

best practices

- code, writing, 665-666
- for pointers, 189-193

bidirectional iterators, 399

binary files, reading, 636-638

binary functions, 512, 521

- binary, 519-520
- lambda expressions, 535-537
- predicates, 522-524

- binary numeral system, 672-673**
 - binary operators**
 - addition/subtraction, 325-327
 - assignment, 327-329
 - types, 323-324
 - binary predicates, 512**
 - elements, removing from linked lists, 458-462
 - elements, sorting in linked lists, 458-462
 - lambda expressions, 537-540
 - binary semaphores, 664**
 - bit flags, 597**
 - bitset class, 598-601
 - vector {bool} class, 603-604
 - bits, 673**
 - bitset class, 598-601**
 - bitwise operators, 92-94**
 - blocks, 79**
 - multiple statements, executing conditionally, 109-110
 - blogs, “C++11 Core Language Feature Support”, 667**
 - bool data type, 37**
 - Boolean values, storing, 37**
 - braces {}, executing multiple statements conditionally, 109-110**
 - break statement, 128-129**
 - bytes, 673**
- C**
- C++11, new features, 12**
 - “C++11 Core Language Feature Support” blog, online documentation, 667**
 - C-style strings, 72**
 - buffer, writing to, 629-630
 - null terminator, analyzing, 70-71
 - calculating Fibonacci numbers with nested loops, 136-137**
 - CALL instruction, 158-159**
 - calling functions, 21-23**
 - calls, 144**
 - capacity of vectors, 436-437**
 - capture lists, maintaining state in lambda expressions, 532-534**
 - case conversion, STL string classes, 418-419**
 - casting operators**
 - const cast, 361-362
 - defined, 353
 - dynamic cast, 357-360
 - need for, 354
 - reinterpret cast, 360
 - static cast, 356
 - troubleshooting, 362-363
 - unpopular styles, 355
 - catch blocks**
 - exception class
 - catching exceptions, 652
 - custom exceptions, throwing, 653-655
 - exception handling, 645-648
 - changing display number formats, 624-627**
 - char data type, 37**
 - characters**
 - arrays, declaring, 406
 - STL string classes, accessing, 410-411
 - values, storing, 38
 - cin statement, 25**
 - classes**
 - abstract base classes, 296-298
 - bitset, 598-601
 - constructors
 - copy constructors, 228-234
 - default parameter values, 219-220
 - initialization lists, 220-222
 - move constructors, 234
 - order of construction, 268
 - overloading, 217-219
 - shallow copying, 226-228
 - container classes, 467**
 - map and multimap, 487
 - multiset container class, deleting elements, 475-480
 - multiset container class, inserting elements, 471-473
 - multiset container class, locating elements, 473-475
 - searching elements, 475
 - set container class, deleting elements, 475-480
 - set container class, inserting elements, 471-473
 - set container class, instantiating, 469-471
 - set container class, locating elements, 473-475
 - declaring, 204**
 - destructors**
 - declaring, 222-223
 - order of destruction, 268-271
 - private destructors, 239-240
 - when to use, 223-225
 - encapsulation, 205**
 - friend classes, declaring, 245-247**
 - inheritance, 252**
 - base class initialization, 258-261
 - multiple inheritance, 277-280
 - private inheritance, 271-273

- protected inheritance, 256-258, 273-276
- public inheritance, 253-254
- syntax, 254-256
- member variables, initializing with constructors, 213-214
- members, accessing
 - with dot operator, 206
 - with pointer operator, 206-208
- private keyword, 208-212
- public keyword, 208-210
- sizeof(), 242-244
- STL deque, 438-440
- STL dynamic array
 - accessing elements in vectors, 431-434
 - deleting elements from vectors, 434-435
 - inserting elements into vectors, 426
 - instantiating vectors, 424-425
 - need for, 423
 - size and capacity of vectors, 436-437
 - vectors, 424
- STL string, 405
 - accessing, 410-411
 - applying, 407
 - case conversion, 418-419
 - concatenation, 412
 - find member function, 413-415
 - instantiating, 407-409
 - need for, 406-407
 - reversing, 417-418
 - template-based implementation, 420
 - truncating, 415-417
- stream classes, 623
- strings, 74
- subclasses, 254
- templates, 382, 385-389
 - protected inheritance, 256-258, 273-276
 - public inheritance, 253-254
 - syntax, 254-256
 - member variables, initializing with constructors, 213-214
 - members, accessing
 - with dot operator, 206
 - with pointer operator, 206-208
 - private keyword, 208-212
 - public keyword, 208-210
 - sizeof(), 242-244
 - STL deque, 438-440
 - STL dynamic array
 - accessing elements in vectors, 431-434
 - deleting elements from vectors, 434-435
 - inserting elements into vectors, 426
 - instantiating vectors, 424-425
 - need for, 423
 - size and capacity of vectors, 436-437
 - vectors, 424
 - STL string, 405
 - accessing, 410-411
 - applying, 407
 - case conversion, 418-419
 - concatenation, 412
 - find member function, 413-415
 - instantiating, 407-409
 - need for, 406-407
 - reversing, 417-418
 - template-based implementation, 420
 - truncating, 415-417
 - stream classes, 623
 - strings, 74
 - subclasses, 254
 - templates, 382, 385-389
 - versus struct keyword, 244-245
- classification of STL algorithms**
 - mutating, 545-547
 - nonmutating, 544-545
- closing files, 632-633**
- code**
 - debugging
 - exception handling with catch blocks, 646-648
 - throwing exceptions, 648-649
 - writing, best practices, 665-666
- code listings**
 - accessing STL strings, 410-411
 - bitset class
 - instantiating, 598-599
 - member methods, 600-601
 - operators, 599-600
 - Calendar class, 314
 - dynamic casting, 358-359
 - functions
 - binary, 519-520
 - unary, 513-514
 - map and multimap, customizing sort predicates, 499-503
 - operators
 - assignment, 328
 - binary, 325-326
 - subscript, 339
 - queues
 - instantiating, 585-586
 - member functions, 587
 - priority, 589-594
 - set and multiset, searching elements, 475
 - simple smart Pointer class, 321-322
 - smart pointers, implementing, 609
- stacks
 - instantiating, 581-582
 - member functions, 582
- STL algorithms
 - copy and remove operations, 562-565
 - counting and finding elements, 550
 - initializing elements, 554-557
 - inserting elements, 572-574
 - partitioning ranges, 570-572
 - processing elements, 557-559
 - replacing elements, 565-567
 - searching ranges, 552
 - sorting collections, 567-570
 - transforming ranges, 560-562
- STL deque class, 440
- STL string classes
 - case conversion, 418-419
 - concatenation, 412
 - find member function, 413-415
 - instantiation, 408-409
 - reversing, 417-418
 - template-based implementation, 420
 - truncating, 415-417
- templates
 - classes, 385-386
 - connecting, 400-401
- vectors
 - accessing elements in, 431-434
 - deleting elements from, 434-435
 - inserting elements into, 426
 - instantiating, 424-425
 - size and capacity of, 436-437

- CodeGuru website, 668**
- collections**
 - elements, inserting, 572-574
 - ranges, searching, 552
 - sorting, 567-570
- collisions, 504**
- commands (preprocessor), #define, 372**
- comments, 21**
- comparing**
 - arrays and pointers, 184-186
 - struct keyword and classes, 244-245
- comparison operators, overloading, 330**
- compile-time checks, performing, 388**
- compilers, determining variable type with auto keyword, 42-44**
- compiling Hello World application, 12**
- components of C++ program, 16**
 - body, 17
 - comments, 21
 - functions, 21-23
 - input, 24-26
 - namespaces, 20
 - output, 24-26
 - preprocessor directive, 16
 - values, returning, 18
 - variables types, 36-38
- compound assignment operators, 96-98**
- compound statements, 79**
- concatenation, STL string classes, 412**
- conditional operator, 118-119**
- conditional statements**
 - if...else, 107-109
 - multiple statements, executing conditionally, 109-110
- connecting STL, 400-402**
- const cast, applying, 361-362**
- const keyword**
 - for pointers, 181-182
 - for references, 196
- constants**
 - defining with #define directive, 50, 368-371
 - enumerated, 48-50
 - literal, 45
 - naming, 51
 - variables, declaring as, 46-47
- constructors**
 - copy constructors, 228-234
 - private copy constructors, 235-236
 - singletons, 236-238
 - declaring, 212-213
 - default constructor, 215
 - default parameter values, 219-220
 - initialization lists, 220-222
 - move constructor, 234, 344
 - order of construction, 268
 - order of destruction, 268-271
 - overloading, 215-219
 - shallow copying, 226-228
 - virtual copy constructors, 304-307
 - when to use, 213-214
- container adapters, 398**
- container classes, 467.**
 - See also containers**
 - advantages of set and multiset, 480
 - elements, searching, 475
 - map and multimap, 487
 - deleting elements, 497-499
 - inserting elements, 491-494
 - instantiating, 489-490
 - locating elements, 494-497
 - sort predicate, customizing, 499-503
- multiset**
 - advantages of, 480-484
 - elements, deleting, 475-480
 - elements, inserting, 471-473
 - elements, locating, 473-475
- set**
 - advantages of, 480-484
 - elements, deleting, 475-480
 - elements, inserting, 471-473
 - elements, locating, 473-475
 - instantiating, 469-471
 - unordered map, 504-508
 - unordered multimap, 504-508
- containers**
 - adaptive, 579-580
 - priority queues, 589-594
 - queues,
 - inserting/removing elements, 587-589
 - queues, instantiating, 585-586
 - queues, member functions, 587
 - stacks,
 - inserting/removing elements, 583-585
 - stacks, instantiating, 581-582
 - stacks, member functions, 582
- algorithms, 547
- elements, initializing, 554-557
- searching, 481
- STL
 - associative, 395

- selecting, 396-398
 - sequential, 394
- continue statement, 128-129**
- controlling infinite loops, 130-132**
- conversion operators, 317-319**
- converting**
 - decimal to binary numeral system, 675-676
 - decimal to hexadecimal numeral system, 676
 - strings, 638-640
- copy assignment operator (=), overloading, 335-338**
- copy constructors, 228-234**
 - private, 235-236
 - singletons, 236-238
 - virtual copy constructors, 304-307
- copy function, 562-565**
- copying**
 - algorithms, 546
 - STL string classes, 407-409
- cores (processor), 660-661**
- counting algorithms, 544**
- counting elements, 550**
- cout statement, 17, 24**
- COW (Copy on Write) smart pointers, 613**
- creating text files, 634**
- custom exceptions, throwing, 648-649, 653-655**
- customizing map and multimap template class predicates, 499-503**

D

- dangling pointers, 189**
- data transaction in threads, 663-664**
- deallocating memory, 175-178**
- debugging**
 - exception handling
 - custom exceptions, throwing, 648-649
 - with catch blocks, 646-648
 - executables, 8
- decimal numeral system, 672**
 - ASCII code values, 724-726
 - converting to binary, 675-676
 - converting to hexadecimal, 676
 - displaying numbers in, 624-627
- declarations**
 - function declarations, 22
 - using namespace, 19-20
- declaring**
 - arrays
 - character arrays, 406
 - multidimensional arrays, 65-66
 - static arrays, 59-60
 - classes, 204
 - friend classes, 245-247
 - constructors, 212-213
 - destructors, 222-223
 - functions, inline, 159-160
 - pointers, 166
 - references, 193-194
 - templates, 379
 - with default parameters, 384
 - with multiple parameters, 383-384
 - variables
 - as constants, 46-47
 - bool type, 37
 - char type, 38
 - floating point types, 40
 - global variables, 35-36
 - memory, accessing, 30-32
 - multiple, 32-33
 - signed integer types, 39
 - type of, substituting, 44-45
 - types of, 36
 - unsigned integer types, 39-40
- decrementing operators, effect on pointers, 179-181**
- deep copy, 228-234, 611-612**
- default constructor, 215**
- default parameters**
 - function values, 147-149
 - templates, declaring, 384
- deference operators, 319-323**
- defining**
 - constants with #define, 50
 - string substitutions, 372
 - templates, 378
 - variables, 30-32
 - reserved words, 52
- delete operator, managing memory consumption, 175-178**
- deleting elements**
 - duplicates, 567-570
 - in linked lists, 453-454, 458-462
 - in map and multimap, 497-499
 - in multiset container class, 475-480
 - in set container class, 475-480
 - in vectors, 434-435
- deque, STL deque class, 438-440**
- dereference operator, accessing pointed data, 170-173**
- derivation**
 - base class initialization, 258-261
 - base class methods
 - invoking, 264-266
 - overriding, 261-263
 - hidden methods, 266-268
 - slicing, 277
 - syntax, 254-256
- destruction order of local exception objects, 650-652**
- destructive copy smart pointers, 614-618**

destructors

- destructors, 222-223
- order of destruction, 268-271
- private destructors, 239-240
- shallow copying, 226-228
- virtual destructors, 288-292
- when to use, 223-225

development, IDEs, 8-9**Diamond Problem, 303****disadvantages of macro functions, 377****displaying simple data types, 628-629****divide operator (/), 80-81****do...while statement, 123-125****documentation, “C++11 Core Language Feature Support” blog, 667****dot operator (.), accessing members, 206****double data type, 37****double precision float, 40****duplicate elements, deleting, 567-570****dynamic arrays, 68-69****dynamic cast, applying, 357-360****dynamic memory allocation, 175-178****E****elements**

- characters, accessing, 410-411
- collections, searching, 552
- counting, 550
- finding, 550
- initializing, 554-557
- inserting, 572-574
- processing, 557-559
- replacing, 565-567
- set or multimultiset, searching, 475

encapsulation, 205**enumerated constants, 48-50****equality operators, 84, 330-332****erase() function, 453-454, 475-480, 497-499****errors, Fence-Post, 64****exception base class, 652-655****exception handling**

- custom exceptions, throwing, 648-649
- exceptions, causes of, 644
- local objects, destruction order, 650-652
- with catch blocks, 645-648

executables, writing, 8**executing multiple statements conditionally, 109-110****F****Fence-Post error, 64****Fibonacci numbers, 47, 136-137****FIFO (first-in-first-out) systems, queues, 580****files**

- binary files, reading, 636-638
- opening and closing, 632-633
- text files
 - creating, 634
 - reading, 635-636

find() function, 473-475, 494-497, 504**find member function, STL string classes, 413-415****flags (bit), 597**

- bitset class, 598-601
- vector bool, 603-604

float data type, 37**floating point variable types, 40****for loops, 125-128****forward iterators, 399****forward_list template class, 462-464****for_each algorithm, 557-559****friend classes, declaring, 245-247****function objects**

- binary, 512, 519-524
- unary, 512-519

function operator, 342-345, 348-349**function prototypes, 144****functions, 21-22**

- arguments, 144
 - passing by reference, 156-157
 - passing to, 196-198
- binary, 519-521
 - lambda expressions, 535-537
 - predicates, 522-524
- CALL instruction, 158-159
- calls, 144
- constructors
 - declaring, 212-213
 - default parameter values, 219-220
 - initialization lists, 220-222
 - move constructors, 234
 - overloading, 215-219
 - shallow copying, 226-228
 - when to use, 213-214

copy(), 562-565**definition, 144****destructors**

- declaring, 222-223
- private destructors, 239-240
- when to use, 223-225

erase(), 453-454, 475-480, 497-499**find(), 473-475, 494-497, 504****inlining, 159-160**

- lambda functions, 161-162
- macro functions
 - advantages of, 377
 - assert(), 376
 - writing, 372-374
- main(), 17
- need for, 142-143
- objects, 511
 - applying, 512, 519-520
 - overview of, 512
- operators. *See* operators
- overloaded, 152-154
- parameters
 - arrays as parameters, 154-155
 - with default values, 147-149
- pointers, passing to, 182-184
- pop_back, 434
- queues, 587, 590-594
- recursive functions, 149-150
- remove(), 562-565
- reverse(), 455-456
- sort(), 456-462
- stacks, 582
- template functions, 379-381
- unary, 512-516
 - lambda expressions, 529-530
 - predicates, 517-519
- values, returning, 18, 23-24
- virtual functions, 292-296
 - polymorphic behavior, implementing, 286-288
- with multiple parameters, 145-146
- with multiple return statements, 151-152
- with no parameters, programming, 146-147
- with no return value, programming, 146

G-H

- generating executables, 8**
- global variables, 35-36**
- goto statement, 119-121**

hash tables

- collisions, 504
- containers, searching in, 481
- unordered map class, 504-508
- unordered multimap class, 504-508

header files, 371

Hello.cpp file, 10

hello world program

- main() function, 17
- preprocessor directive, 16
- source code, 10

hexadecimal numeral system, 674

- ASCII code values, 724-726
- displaying integers in, 624-627

hidden methods, 266-268

history of C++, 7

I

I-values, 80

IDEs (Integrated Development Environments), 8-9

if...else statements, 107-109

- nested if statements, 111-114

implementing

- constructors, 212-214
- destructors, 222-225
- smart pointers, 609-610

include statement, 16

increment operator (++), 81, 179-181

inequality operators, 330-332

infinite loops, 129-132

inheritance, 252

- base class methods
 - initialization, 258-261
 - invoking in derived class, 264-266
 - overriding, 261-263
- hidden methods, 266-268
- multiple inheritance, 277-280
- order of construction, 268
- order of destruction, 268-271
- overridden methods, invoking, 263
- polymorphism, 284-285
 - abstract base classes, 296-298
 - implementing with virtual functions, 286-288
 - virtual functions, 292-296
- private inheritance, 271-273
- protected inheritance, 256-258, 273-276
- public inheritance, 253-254
- slicing, 277
- subclasses, 254
- syntax, 254-256
- virtual inheritance, 299-303

initialization algorithms, 545

initialization lists, 220-222

- base class initialization, 259-261

initializing

- arrays, static arrays, 59-60
- class member variables via constructors, 213-214
- elements, 554-557
- lists, 446-447
- multidimensional arrays, 65-66
- variables, 31-33

inline functions, 159-160

input, 24-26

input iterators, 399

inserting

- elements, 572-574

- in linked lists, 448-453
- in map and multimap template classes, 491-494
- in multiset container class, 471-473
- in set container class, 471-473
- in singly-linked lists, 462-464
- in vectors, 426
- queue elements, 587-589
- stack elements, 583-585
- text into strings, 630-632
- instantiating**
 - bitset classes, 598-599
 - map template class, 489-490
 - queues, 585-590
 - set objects, 469-471
 - stacks, 581-582
 - STL string classes, 407-409
 - templates, 383
 - vector {bool} class, 603-604
 - vectors, 424-425
- int data type, 37**
- integers**
 - signed, 38-39
 - size of, determining, 40-42
 - unsigned, 38-40
- Intel 8086 processor, 660**
- intrusive reference counting, 613**
- invalid pointers, 187-188**
- invoking**
 - base class methods in derived class, 264-266
 - overridden methods, 263
- iterating multidimensional arrays with nested loops, 134-135**
- iterators**
 - STL, 399
 - vector elements, accessing, 433-434
- J-K**
- junk value, 166**
- key-value pairs, hash tables**
 - collisions, 504
 - unordered map class, 504-508
 - unordered multimap class, 504-508
- keywords, 52, 677-678**
 - auto, 42-44
 - const, 196
 - private, 210-212
 - protected inheritance, 256-258
 - struct, 244-245
- L**
- lambda expressions, 515**
 - for binary functions, 535-537
 - for binary predicates, 537-540
 - state, maintaining, 532-534
 - syntax, 534-535
 - for unary functions, 529-530
 - for unary predicates, 531-532
- lambda functions, 161-162**
- libraries, smart pointers, 618**
- LIFO (last-in-first-out) systems, stacks, 580**
- linked lists, 445. See also lists**
 - list template class
 - characteristics of, 446
 - elements, erasing, 453-454, 458-462
 - elements, inserting, 448-453
 - elements, reversing, 455-456
 - elements, sorting, 456-462
 - instantiating, 446-447
 - singly-linked lists, 462-464
- list template class**
 - elements
 - erasing, 453-454, 458-462
 - inserting, 448-453
 - reversing, 455-456
 - sorting, 456-462
 - instantiating, 446-447
- lists**
 - captures lists, maintaining state in lambda expressions, 532-534
 - elements
 - erasing, 453-454, 458-462
 - inserting, 448-453
 - reversing, 455-456
 - sorting, 456-462
 - initializing, 446-447
 - singly-linked, 462-464
- literal constants, 45**
- locating elements**
 - in map template class, 494-496
 - in multimap template class, 496-497
 - in multiset container class, 473-475
 - in set container class, 473-475
- logical operators, 87-92**
- long int data type, 37**
- loops**
 - break statement, 128-129
 - continue statement, 128-129
 - do...while statement, 123-125
 - for loops, 125-128
 - goto statement, 119-121
 - infinite loops, 129-132
 - nested loops, 133

- Fibonacci numbers,
 - calculating, 136-137
- multidimensional arrays,
 - iterating, 134-135
- while statement, 121-123

M

macro functions

- advantages of, 377
- syntax, 374-375
- writing, 372-374

macros

- #define, defining constants, 368-371
- assert(), validating expressions, 376
- multiple inclusion, preventing, 371-372

main() function, 17

maintaining state in lambda expressions, 532-534

managing memory consumption. *See also* memory; smart pointers

- delete operator, 175-178
- new operator, 175-178

manipulating strings, 72-74

map template class

- elements
 - deleting, 497-499
 - inserting, 491-494
 - locating, 494-496
- instantiating, 489-490
- sort predicate, customizing, 499-503

member functions

- queues, 587, 590-594
- stacks, 582

member methods, bitset classes, 600-601

memory, 30

- accessing with variables, 30-32

arrays

- organization in, 60
- zero-based index, 61-62
- CALL instruction, 158-159
- of classes, sizeof(), 242-244
- deep copying, 228-234
- dynamic allocation
 - delete operator, 175-178
 - new operator, 175-178
- I-values, 80
- invalid memory locations, pointing to, 187-188
- leaks, 187
- pointers
 - declaring, 166
 - smart pointers, 608-618
 - this pointer, 241
- shallow copying, 226-228
- variables, determining size of, 40-42

methods

- base class, overriding, 261-263
- bitset class, 600-601
- hidden, 266-268
- overridden, invoking, 263
- push back(), 426, 448-453
- push front(), 448-453

microprocessors, CALL instruction, 158-159

modifying

- algorithms, 546
- data in arrays, 62-65

modulo divide operator (%), 80-81

move constructors, 234, 344

multicore processors, 660-661

- threads
 - data transaction, 663-664
 - synchronization, 664

multidimensional arrays

- accessing elements in, 66-68
- declaring, 65-66

multimap template class, 487

elements

- deleting, 497-499
- inserting, 491-494
- locating, 496-497
- instantiating, 489-490
- sort predicate, customizing, 499-503

multiple inclusion, preventing with macros, 371-372

multiple inheritance, 254, 277-280

multiple parameters

- functions, programming, 145-146
- templates, declaring, 383-384

multiple return statements for functions, 151-152

multiple variables, declaring, 32-33

multiply operator (*), 80-81

multiset template class, 467

- advantages of, 480-484

elements

- deleting, 475-480
- inserting, 471-473
- locating, 473-475

multithreaded applications, 661-662

- problems caused by, 664-665
- programming, reasons for, 662-663

mutating algorithms, 545-547

mutexes, thread synchronization, 664

N

namespaces, 19-20

naming

- constants, 51
- variables, 32, 51-52, 677-678

nested if statements, 111-114

nested loops, 133

- Fibonacci numbers,
 - calculating, 136-137
- multidimensional arrays,
 - iterating, 134-135

new features in C++11, 12

new operator, managing memory consumption, 175-178

non-redefinable operators, 349-350

nonmutating algorithms, 544-545

NOT operator, 87-94

nt data type, 37

null terminator, analyzing, 70-71

number formats, changing, 624-627

O

objects, function objects, 511

applying, 512, 519-520

binary functions, 519-524

overview of, 512

unary functions, 512-519

online documentation, "C++11 Core Language Feature Support" blog, 667

opening files, 632-633

operators

AND operator (&), determining variable address, 167-168

add operator (+), 80-81

assignment operator (=), 79

binary

addition/subtraction, 325-327

assignment, 327-329

types, 323-324

bitset classes, 599-600

bitwise, 92-94

casting

const cast, 361-362

defined, 353

dynamic cast, 357-360

need for, 354

reinterpret cast, 360

static cast, 356

troubleshooting, 362-363

unpopular styles, 355

comparison operators, overloading, 330

compound assignment operators, 96-98

conditional operators, 118-119

copy assignment (=) operator, overloading, 335-338

decrement operator (—), 81

divide operator (/), 80-81

dot operator, accessing members, 206

equality operators, 84, 330-332

function operators, 342-349

increment (++) operator, 81

inequality operator, 330-332

logical operators, 87-92

modulo divide operator (%), 80-81

multiply operator (*), 80-81

non-redefinable, 349-350

OR operator (||), 88-92

overloading, 332-335, 342

pointer operator, accessing members, 206-208

postfix, 81-84

precedence, 99-101, 679-680

prefix, 81-84

relational operators, 85-87

sizeof(), 98-99

stream extraction operator, 622

subscript, 338-341

subtract operator (-), 80-81

symbols, 312

types, 312-313

unary

conversion operators, 317-319

decrement operators, 314-317

increment operators, 314-317

programming deference, 319-323

types, 313

OR operator (||), 88-92

order of construction, 268

order of destruction, 268-271

organization of arrays, 60

output, 24-26

output iterators, 399

overflows, 83

overloaded functions, 152-154

overloading

binary operators, 324

comparison operators, 330

constructors, 215-219

copy assignment operator, 335-338

hidden methods, 266-268

non-redefinable operators, 349-350

operators, 313, 332-335, 342

overridden methods, invoking, 263

P

parameters

for functions

with default values, 147-149

arrays as parameters, 154-155

multiple parameters, programming, 145-146

templates, declaring, 383-384

partitioning

- algorithms, 547
- ranges, 570-572

passing arguments by reference, 156-157**performance**

- multicore processors, 660-661
- multithreaded applications, 661
- problems caused by, 664-665
- programming, reasons for, 662-663

performing compile-time asserts, 388**pointed data, accessing with dereference operator, 170-173****pointer operator (->), 206-208****pointers**

- addresses, storing, 168-170
- arrays, similarity to, 184-186
- best practices, 189-193
- const keyword, 181-182
- dangling pointers, 189
- declaring, 166
- incrementing/decrementing operators, effect of, 179-181
- invalid memory locations, 187-188
- memory leaks, 187
- passing to functions, 182-184
- size of, 173-174
- smart pointers, 607-608
 - COW, 613
 - deep copy, 611-612
 - destructive copy, 614-618
 - implementing, 609-610
 - libraries, 618
 - reference counted, 613-614
 - reference-linked, 614
- this pointer, 241

polymorphism

- abstract base classes, 296-298
- implementing with virtual functions, 286-288
- need for, 284-286
- virtual copy constructors, 304-307
- virtual functions, 292-296
- virtual inheritance, 299-303

pop operation, 158**pop() function, 583-589****pop_back() function, 434****postfix operators, 81-84****precedence, operator precedence, 99-101, 679-680****predicates**

- binary, 512, 522-524
 - elements, removing from linked lists, 458-462
 - elements, sorting in linked lists, 458-462
 - lambda expressions, 537-540
- unary, 517-519, 531-532

prefix operators, 81-84**preprocessor directives, 16**

- # define, defining constants, 50, 368-372

macro functions, writing, 372-374**preventing multiple inclusion with macros, 371-372****priority queues, 589-594****private copy constructors, 235-236****private destructors, 239-240****private inheritance, 271-273****private keyword, 208-212****processing elements, 557, 559****processors**

- cores, 660-661

Intel 8086, 660

multithreaded applications, 661

data transaction, 663-664

problems caused by, 664-665

reasons for programming, 662-663

thread synchronization, 664

program flow, controlling

- if...else statement, 107-109
- nested if statements, 111-114
- switch-case statement, 115-117

programming

- templates, 389
 - connecting, 400-402
 - STL algorithms, 400
 - STL containers, 394-398
 - STL iterators, 399
- unary operators, 319-323

properties of STL container classes, 396**protected inheritance, 256-258, 273-276****public inheritance, 253-254****public keyword, 208-210****push() function, 583-589****push back() method, 426, 448-453****push front() method, 448-453****push operation, 158****Q-R****queues, 580-581**

- inserting/removing elements, 587-589
- instantiating, 585-586

RAM, 30**random access iterators, 399****ranges**

- elements, processing, 557-559
- partitioning, 570-572
- searching, 552

- transforming, 560-562
- values, replacing, 565-567
- reading**
 - binary files, 636-638
 - text files, 635-636
- recursion, preventing multiple inclusion with macros, 371-372**
- recursive functions, 149-150**
- reference-counted smart pointers, 613-614**
- reference-linked smart pointers, 614**
- references, 193**
 - arguments, passing to functions, 196-198
 - const keyword, 196
 - utility of, 194-196
- reinterpret cast, applying, 360**
- relational operators, 85-87**
- removal algorithms, 546**
- remove function, 562-565**
- removing**
 - elements from singly-linked lists, 462-464
 - queue elements, 587-589
 - stack elements, 583-585
- repeating code, loops**
 - break statement, 128-129
 - continue statement, 128-129
 - do...while statement, 123-125
 - for statement, 125-128
 - goto statement, 119-121
 - infinite loops, 129-132
 - nested loops, 133-134
 - while statement, 121-123
- replacement algorithms, 546**
- replacing elements, 565-567**
- reserved words, 52, 677-678**
- return statements, multiple, 151-152**
- returning values, 18, 23-24**
- reverse() function, 455-456**
- reversing**
 - elements in linked lists, 455-456
 - STL string classes, 417-418
- RTTI (Run Time Type Identification), 296**
- runtime type identification, 357-360**
- S**
 - scientific notation, displaying integers in, 626-627**
 - scope of variables, 33-34**
 - sdefine (#define) statement, string substitutions, 372**
 - searching**
 - algorithms, 544
 - elements, 550
 - in map template classes, 494-496
 - in multimap template class, 496-497
 - set or multiset, 475
 - in containers, 481
 - ranges, 552
 - selecting containers, 396-398**
 - semaphores, thread synchronization, 664**
 - sequential containers, 394**
 - set template class, 467**
 - advantages of, 480-484
 - elements
 - deleting, 475-480
 - inserting, 471-473
 - locating, 473-475
 - objects, instantiating, 469-471
 - shallow copying, 226-228**
 - short int data type, 37**
 - sign-bits, 38**
 - signed integer types, 39**
 - signed integers, 38**
- simple data types, displaying, 628-629**
- singletons, 236-238**
- singly-linked lists, 462-464**
- sizeof(), 40-42, 98-99**
 - for classes, determining, 242-244
 - pointers, 173-174
- sizing vectors, 436-437**
- slicing, 277**
- smart pointers, 607**
 - COW, 613
 - deep copy, 611-612
 - destructive copy, 614-618
 - implementing, 609-610
 - libraries, 618
 - overview of, 608
 - reference counted, 613-614
 - reference-linked, 614
- sort predicate (map/multimap template classes), customizing, 499-503**
- sort() function, 456-462**
- sorting**
 - algorithms, 546
 - collections, 567-570
 - elements in linked lists, 456-462
- specialization, templates, 383**
- stacks, 580**
 - inserting/removing elements, 583-585
 - instantiating, 581-582
 - local exception object
 - destruction order, 650-652
 - member functions, 582
 - operations, 158
- state, maintaining in lambda expressions, 532-534**
- statements**
 - #define, string substitutions, 372
 - break, 128-129
 - compound, 79

- conditional statements,
 - if...else, 107-109
- continue, 128-129
- cout, 17
- do...while, 123-125
- for, 125-128
- goto, 119-121
- nested if statements, 111-114
- switch-case, 115-117
- syntax, 78
- while, 121-123
- static arrays, declaring, 59-60**
- static cast, applying, 356**
- static members of template classes, 386-389**
- std namespace, 19-20**
- std::string, 72-74**
- STL (Standard Template Library), 389**
 - adaptive containers, 579
 - instantiating queues, 585-586
 - instantiating stacks, 581-582
 - priority queues, 589-594
 - queue member functions, 587
 - queues, 580-581
 - queues,
 - inserting/removing elements, 587-589
 - stack member functions, 582
 - stack, inserting/removing elements, 583-585
 - stacks, 580
- algorithms, 400, 543
 - classification of, 544-547
 - copy and remove operations, 562-565
 - counting and finding elements, 550
 - initializing elements, 554-557
 - inserting elements, 572-574
 - overview of, 544
 - partitioning ranges, 570-572
 - processing elements, 557-559
 - replacing elements, 565-567
 - searching ranges, 552
 - sorting collections, 567-570
 - transforming ranges, 560-562
- bit flags, 597-601
- connecting, 400-402
- container adapters, 398
- container classes, 467
 - advantages of set and multiset, 480
 - multiset, 480-484
 - searching elements, 475
 - set, 480-484
- containers
 - associative, 395
 - selecting, 396-398
 - sequential, 394
- deque class, 438-440
- dynamic array class
 - accessing elements, 431-434
 - deleting elements, 434-435
 - inserting elements, 426
 - instantiating, 424-425
 - need for, 423
 - size and capacity, 436-437
 - vectors, 424
- exception base class
 - catching exceptions, 652
 - custom exceptions,
 - throwing, 653-655
- forward_list template class, 462-464
- iterators, 399
- linked lists, 445
- list template class
 - elements, erasing, 453-454, 458-462
 - elements, inserting, 448-453
 - elements, reversing, 455-456
 - elements, sorting, 456-462
 - instantiating, 446-447
- map and multimap, 487
 - deleting elements, 497-499
 - inserting elements, 491-494
 - instantiating, 489-490
 - locating elements, 494-497
 - sort predicate,
 - customizing, 499-503
- multiset template class,
 - elements
 - deleting, 475-480
 - inserting, 471-473
 - locating, 473-475
- set template class
 - elements, deleting, 475-480
 - elements, inserting, 471-473
 - elements, locating, 473-475
 - instantiating, 469-471
- string class, 405
 - accessing, 410-411
 - applying, 407
 - case conversion, 418-419
 - concatenation, 412
 - find member function, 413-415
 - instantiating, 407-409
 - need for, 406-407
 - reversing, 417-418
 - template-based implementation, 420
 - truncating, 415-417

- unordered map class, 504-508
- unordered multimap class, 504-508
- vector {bool} class, 603-604
- storing**
 - addresses with pointers, 168-170
 - Boolean values, 37
 - character values, 38
- stray pointers, 189**
- stream classes, 623**
- stream extraction operator, 622**
- streams, 18, 621**
 - binary files, reading, 636-638
 - files, opening and closing, 632-633
 - text files, creating, 634
 - text files, reading, 635-636
- string classes, 74**
- string literals, 18**
 - C-style strings, 70-72
 - null terminator, analyzing, 70-71
- strings**
 - conversion operations, 638-640
 - manipulating, 72-74
 - STL string class, 405
 - accessing, 410-411
 - applying, 407
 - case conversion, 418-419
 - concatenation, 412
 - find member function, 413-415
 - instantiating, 407-409
 - need for, 406-407
 - reversing, 417-418
 - template-based implementation, 420
 - truncating, 415-417
 - substitutions, 372
 - text, inserting, 630-632
- struct keyword, 244-245**
- styles, unpopular casting, 355**
- subclasses, 254**
- subscript operators ([]), 338-341**
- substituting types of variables, 44-45**
- subtract operator (-), 80-81, 325-327**
- subtraction assignment operator (-=), 327-329**
- switch-case statement, 115-117**
- synchronization (threads), 664**
- syntax**
 - inheritance, 254-256
 - lambda expressions, 534-535
 - macros, 374-375
 - statements, 78
 - templates, 378
- element, deleting, 475-480**
- element, inserting, 471-473**
- element, locating, 473-475**
- set container class**
 - advantages of, 480-484
 - element, inserting, 471-473
 - element, locating, 473-475
 - elements, deleting, 475-480
 - instantiating, 469-471
 - vector {bool} class, 603-604
- template functions, 379-381**
- templates**
 - adaptive container, 579
 - instantiating stacks, 581-582
 - priority queues, 589-594
 - queues, instantiating, 580-581, 585-586
 - queues, member functions, 587-589
 - stacks,
 - inserting/removing elements, 583-585
 - stacks, member functions, 580-582
- algorithms, 543**
 - classification of, 544-547
 - copy and remove operations, 562-565
 - counting and finding elements, 550
 - initializing elements, 554-557
 - inserting elements, 572-574
 - overview of, 544
 - partitioning ranges, 570-572

- processing elements, 557-559
 - replacing elements, 565-567
 - searching ranges, 552
 - sorting collections, 567-570
 - transforming ranges, 560-562
 - applying, 389
 - bit flags, 597-601
 - classes, 382, 385-389
 - container classes, 467
 - advantages of set and multiset, 480
 - map and multimap, 487
 - searching elements, 475
 - default parameters, declaring with, 384
 - defining, 378
 - instantiating, 383
 - linked lists, 445
 - multiple parameters, declaring with, 383-384
 - overview of, 378
 - specialization, 383
 - STL
 - algorithms, 400
 - connecting, 400-402
 - containers, 394-398
 - iterators, 399
 - string classes, 420
 - types, declaring, 379
 - ternary operator, 118-119**
 - text**
 - aligning, 627-628
 - inserting into strings, 630-632
 - text files**
 - creating, 634
 - reading, 635-636
 - this pointer, 241**
 - threads**
 - data transaction, 663-664
 - multithreaded applications, 661
 - problems caused by, 664-665
 - reasons for programming, 662-663
 - synchronization, 663-664
 - throwing custom exceptions, 648-649, 653-655**
 - transforming, 560-562**
 - troubleshooting**
 - casting operators, 362-363
 - compiling errors, 448-450
 - truncating STL string classes, 415-417**
 - try blocks, exception handling, 645-647**
 - types**
 - of operators, 312-313
 - binary, 323-329
 - programming deference, 319-323
 - subscript, 338-341
 - unary, 313-319
 - runtime identification, 357-360
 - of STL algorithms
 - mutating, 545-547
 - nonmutating, 544-545
 - of templates declaring, 379
 - of variables, 36
 - bool, 37
 - char, 38
 - determining with auto keyword, 42-44
 - substituting, 44-45
 - U**
 - unary functions, 512-516**
 - lambda expressions, 529-530
 - predicates, 517-519
 - unary operators**
 - conversion operators, 317-319
 - decrement operators, 314-317
 - increment operators, 314-317
 - programming, 319-323
 - types, 313
 - unary predicates in lambda expressions, 531-532**
 - unhandled exceptions, 644**
 - unordered map class, 504-508**
 - unordered multimap class, 504-508**
 - unpopular casting styles, 355**
 - unsigned integer types, 37-40**
 - unsigned long int data type, 37**
 - unsigned short int data type, 37**
 - using namespace declaration, 19-20**
- V**
- validating expressions with assert() macro, 376**
 - values**
 - containers, initializing elements, 554-557
 - replacing, 565-567
 - returning, 18, 23-24
 - variables**
 - auto keyword, 42-44
 - bool type, 37
 - char type, 38
 - declaring as constants, 46-47
 - destructors, virtual destructors, 288-292
 - global, 35-36
 - initializing, 31
 - memory
 - accessing, 30-32
 - address, determining, 167-168
 - usage, determining, 98-99
 - multiple, declaring, 32-33

- names, reserved words, 52, 677-678
 - naming, 32, 51
 - pointers
 - addresses, storing, 168-170
 - arrays, similarity to, 184-186
 - best practices, 189-193
 - const keyword, 181-182
 - dangling pointers, 189
 - declaring, 166
 - incrementing/
decrementing
operators, effect
on, 179-181
 - invalid memory
locations, 187-188
 - memory leaks, 187
 - passing to functions,
182-184
 - size of, 173-174
 - this pointer, 241
 - references, 193
 - arguments, passing to
functions, 196-198
 - const keyword, 196
 - utility of, 194-196
 - scope, 33-34
 - size of, determining, 40-42
 - types, 36
 - floating point types, 40
 - signed integer types, 39
 - substituting, 44-45
 - unsigned integer types,
39-40
 - vector {bool} class**
 - instantiating, 603-604
 - operators, 604
 - vectors**
 - characteristics of, 424
 - elements
 - accessing, 431-434
 - deleting, 434-435
 - inserting, 426
 - instantiating, 424-425
 - size and capacity of, 436-437
 - virtual copy constructors, 304-307**
 - virtual destructors, 288-292**
 - virtual functions, 292-296**
 - polymorphic behavior,
implementing, 286-288
 - virtual inheritance, 299-303**
- ## W
- websites**
 - “C++11 Core Language
Feature Support” blog, 667
 - CodeGuru, 668
 - MSDN, online
documentation, 667
 - while statement, 121-123**
 - width of fields, setting, 627-628**
 - wild pointers, 189**
 - writing**
 - code, best practices, 665-666
 - executables, 8
 - Hello World application,
9-11
 - macro functions, 372-374
 - to binary files, 636-638
 - to C-style string buffer,
629-630
- ## X-Y-Z
- XOR operator, 87-88**
 - zero-based index, 61-62**