

Siddhartha Rao

EIGHTH
EDITION

New **C++14**
& **C++17**
Coverage

Sams **Teach Yourself**

C++

in **One Hour** a Day

SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



Siddhartha Rao

Sams**TeachYourself**

C++

in **One Hour** a Day

Eighth Edition

SAMS

800 East 96th Street, Indianapolis, Indiana 46240 USA

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

Copyright © 2017 by Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/. 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: 978-0-7897-5774-6

ISBN-10: 0-7897-5774-5

Library of Congress Control Number: 2016958138

First Printing: December 2016

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.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Editor

Mark Taber

Senior Project Editor

Tonya Simpson

Copy Editor

Geneil Breeze

Indexer

Erika Millen

Proofreader

Sasirekha Durairajan

Technical Editor

Adrian Ngo

Compositor

codeMantra

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	7
Analyzing Errors and “Debugging”	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++?	12

LESSON 2: The Anatomy of a C++ Program	17
---	-----------

Parts of the Hello World Program	18
Preprocessor Directive <code>#include</code>	18
The Body of Your Program <code>main()</code>	19
Returning a Value	20
The Concept of Namespaces	21
Comments in C++ Code	22
Functions in C++	23
Basic Input Using <code>std::cin</code> and Output Using <code>std::cout</code>	26

LESSON 3: Using Variables, Declaring Constants	31
---	-----------

What Is a Variable?	32
Memory and Addressing in Brief	32
Declaring Variables to Access and Use Memory	32

Declaring and Initializing Multiple Variables of a Type	34
Understanding the Scope of a Variable	35
Global Variables	37
Naming Conventions	38
Common Compiler-Supported C++ Variable Types	39
Using Type <code>bool</code> to Store Boolean Values	40
Using Type <code>char</code> to Store Character Values	41
The Concept of Signed and Unsigned Integers	41
Signed Integer Types <code>short</code> , <code>int</code> , <code>long</code> , and <code>long long</code>	42
Unsigned Integer Types <code>unsigned short</code> , <code>unsigned int</code> , <code>unsigned long</code> , and <code>unsigned long long</code>	42
Avoid Overflow Errors by Selecting Correct Data Types	43
Floating-Point Types <code>float</code> and <code>double</code>	45
Determining the Size of a Variable Using <code>sizeof</code>	46
Avoid Narrowing Conversion Errors by Using List Initialization	48
Automatic Type Inference Using <code>auto</code>	48
Using <code>typedef</code> to Substitute a Variable's Type	50
What Is a Constant?	50
Literal Constants	51
Declaring Variables as Constants Using <code>const</code>	52
Constant Expressions Using <code>constexpr</code>	53
Enumerations	55
Defining Constants Using <code>#define</code>	57
Keywords You Cannot Use as Variable or Constant Names	58
LESSON 4: Managing Arrays and Strings	63
What Is an Array?	64
The Need for Arrays	64
Declaring and Initializing Static Arrays	65
How Data Is Stored in an Array	66
Accessing Data Stored in an Array	67
Modifying Data Stored in an Array	69
Multidimensional Arrays	71
Declaring and Initializing Multidimensional Arrays	72
Accessing Elements in a Multidimensional Array	73

Dynamic Arrays	74
C-style Character Strings	76
C++ Strings: Using <code>std::string</code>	79
LESSON 5: Working with Expressions, Statements, and Operators	85
Statements	86
Compound Statements or Blocks	87
Using Operators	87
The Assignment Operator (=)	87
Understanding L-values and R-values	87
Operators to Add (+), Subtract (-), Multiply (*), Divide (/), and Modulo Divide (%)	88
Operators to Increment (++) and Decrement (--)	89
To Postfix or to Prefix?	90
Equality Operators (==) and (!=)	92
Relational Operators	92
Logical Operations NOT, AND, OR, and XOR	95
Using C++ Logical Operators NOT (!), AND (&&), and OR ()	96
Bitwise NOT (~), AND (&), OR (), and XOR (^) Operators	100
Bitwise Right Shift (>>) and Left Shift (<<) Operators	102
Compound Assignment Operators	104
Using Operator <code>sizeof</code> to Determine the Memory Occupied by a Variable	106
Operator Precedence	108
LESSON 6: Controlling Program Flow	113
Conditional Execution Using <code>if ... else</code>	114
Conditional Programming Using <code>if ... else</code>	115
Executing Multiple Statements Conditionally	117
Nested <code>if</code> Statements	118
Conditional Processing Using <code>switch-case</code>	122
Conditional Execution Using Operator (<code>?:</code>)	126
Getting Code to Execute in Loops	128
A Rudimentary Loop Using <code>goto</code>	128
The <code>while</code> Loop	130
The <code>do...while</code> Loop	132
The <code>for</code> Loop	133
The Range-Based <code>for</code> Loop	137

Modifying Loop Behavior Using <code>continue</code> and <code>break</code>	139
Loops That Don't End—That Is, Infinite Loops	140
Controlling Infinite Loops	141
Programming Nested Loops	143
Using Nested Loops to Walk a Multidimensional Array	145
Using Nested Loops to Calculate Fibonacci Numbers	147
LESSON 7: Organizing Code with Functions	151
The Need for Functions	152
What Is a Function Prototype?	153
What Is a Function Definition?	154
What Is a Function Call, and What Are Arguments?	154
Programming a Function with Multiple Parameters	155
Programming Functions with No Parameters or No Return Values	156
Function Parameters with Default Values	157
Recursion—Functions That Invoke Themselves	159
Functions with Multiple Return Statements	161
Using Functions to Work with Different Forms of Data	162
Overloading Functions	163
Passing an Array of Values to a Function	165
Passing Arguments by Reference	166
How Function Calls Are Handled by the Microprocessor	168
Inline Functions	169
Automatic Return Type Deduction	171
Lambda Functions	172
LESSON 8: Pointers and References Explained	177
What Is a Pointer?	178
Declaring a Pointer	178
Determining the Address of a Variable Using the Reference Operator (<code>&</code>)	179
Using Pointers to Store Addresses	180
Access Pointed Data Using the Dereference Operator (<code>*</code>)	183
What Is the <code>sizeof()</code> of a Pointer?	185
Dynamic Memory Allocation	187
Using Operators <code>new</code> and <code>delete</code> to Allocate and Release Memory Dynamically	187
Effect of Incrementing and Decrementing Operators (<code>++</code> and <code>--</code>) on Pointers	191

Using the <code>const</code> Keyword on Pointers	193
Passing Pointers to Functions	194
Similarities between Arrays and Pointers	195
Common Programming Mistakes When Using Pointers	198
Memory Leaks	198
When Pointers Don't Point to Valid Memory Locations	199
Dangling Pointers (Also Called Stray or Wild Pointers)	200
Checking Whether Allocation Request Using <code>new</code> Succeeded	202
Pointer Programming Best-Practices	204
What Is a Reference?	205
What Makes References Useful?	206
Using Keyword <code>const</code> on References	208
Passing Arguments by Reference to Functions	208

PART II: Fundamentals of Object-Oriented C++ Programming

LESSON 9: Classes and Objects 215

The Concept of Classes and Objects	216
Declaring a Class	216
An Object as an Instance of a Class	217
Accessing Members Using the Dot Operator (.)	218
Accessing Members Using the Pointer Operator (->)	219
Keywords <code>public</code> and <code>private</code>	220
Abstraction of Data via Keyword <code>private</code>	222
Constructors	224
Declaring and Implementing a Constructor	224
When and How to Use Constructors	225
Overloading Constructors	227
Class Without a Default Constructor	228
Constructor Parameters with Default Values	230
Constructors with Initialization Lists	231
Destructor	233
Declaring and Implementing a Destructor	234
When and How to Use a Destructor	234
Copy Constructor	237
Shallow Copying and Associated Problems	237
Ensuring Deep Copy Using a Copy Constructor	240
Move Constructors Help Improve Performance	244

Different Uses of Constructors and the Destructor	246
Class That Does Not Permit Copying	246
Singleton Class That Permits a Single Instance	247
Class That Prohibits Instantiation on the Stack	249
Using Constructors to Convert Types	251
this Pointer	254
sizeof () a Class	255
How struct Differs from class	257
Declaring a friend of a class	258
union: A Special Data Storage Mechanism	260
Declaring a Union	260
Where Would You Use a union?	261
Using Aggregate Initialization on Classes and Structs	263
constexpr with Classes and Objects	266
LESSON 10: Implementing Inheritance	271
Basics of Inheritance	272
Inheritance and Derivation	272
C++ Syntax of Derivation	274
Access Specifier Keyword protected	276
Base Class Initialization—Passing Parameters to the Base Class	279
Derived Class Overriding Base Class's Methods	281
Invoking Overridden Methods of a Base Class	283
Invoking Methods of a Base Class in a Derived Class	284
Derived Class Hiding Base Class's Methods	286
Order of Construction	288
Order of Destruction	288
Private Inheritance	291
Protected Inheritance	293
The Problem of Slicing	297
Multiple Inheritance	297
Avoiding Inheritance Using final	300
LESSON 11: Polymorphism	305
Basics of Polymorphism	306
Need for Polymorphic Behavior	306
Polymorphic Behavior Implemented Using Virtual Functions	308

Need for Virtual Destructors	310
How Do virtual Functions Work? Understanding the Virtual Function Table	314
Abstract Base Classes and Pure Virtual Functions	318
Using virtual Inheritance to Solve the Diamond Problem	321
Specifier <code>override</code> to Indicate Intention to Override	326
Use <code>final</code> to Prevent Function Overriding	327
Virtual Copy Constructors?	328
LESSON 12: Operator Types and Operator Overloading	335
What Are Operators in C++?	336
Unary Operators	337
Types of Unary Operators	337
Programming a Unary Increment/Decrement Operator	338
Programming Conversion Operators	341
Programming Dereference Operator (*) and Member Selection Operator (->)	344
Binary Operators	346
Types of Binary Operators	346
Programming Binary Addition (a+b) and Subtraction (a-b) Operators	347
Implementing Addition Assignment (+=) and Subtraction Assignment (-=) Operators	350
Overloading Equality (==) and Inequality (!=) Operators	352
Overloading <, >, <=, and >= Operators	354
Overloading Copy Assignment Operator (=)	357
Subscript Operator ([])	360
Function Operator ()	364
Move Constructor and Move Assignment Operator for High Performance Programming	365
The Problem of Unwanted Copy Steps	365
Declaring a Move Constructor and Move Assignment Operator	366
User Defined Literals	371
Operators That Cannot Be Overloaded	373
LESSON 13: Casting Operators	377
The Need for Casting	378
Why C-Style Casts Are Not Popular with Some C++ Programmers	379

The C++ Casting Operators	379
Using <code>static_cast</code>	380
Using <code>dynamic_cast</code> and Runtime Type Identification	381
Using <code>reinterpret_cast</code>	384
Using <code>const_cast</code>	385
Problems with the C++ Casting Operators	386
LESSON 14: An Introduction to Macros and Templates	391
The Preprocessor and the Compiler	392
Using Macro <code>#define</code> to Define Constants	392
Using Macros for Protection against Multiple Inclusion	395
Using <code>#define</code> to Write Macro Functions	396
Why All the Parentheses?	398
Using Macro <code>assert</code> to Validate Expressions	399
Advantages and Disadvantages of Using Macro Functions	400
An Introduction to Templates	402
Template Declaration Syntax	402
The Different Types of Template Declarations	403
Template Functions	403
Templates and Type Safety	405
Template Classes	406
Declaring Templates with Multiple Parameters	407
Declaring Templates with Default Parameters	408
Sample Template <code>class<> HoldsPair</code>	408
Template Instantiation and Specialization	410
Template Classes and <code>static</code> Members	412
Variable Templates, Also Called Variadic Templates	413
Using <code>static_assert</code> to Perform Compile-Time Checks	417
Using Templates in Practical C++ Programming	418
PART III: Learning the Standard Template Library (STL)	
LESSON 15: An Introduction to the Standard Template Library	421
STL Containers	422
Sequential Containers	422
Associative Containers	423
Container Adapters	425

STL Iterators	425
STL Algorithms	426
The Interaction between Containers and Algorithms Using Iterators	427
Using Keyword <code>auto</code> to Let Compiler Define Type	429
Choosing the Right Container	429
STL String Classes	432
LESSON 16: The STL String Class	435
The Need for String Manipulation Classes	436
Working with the STL String Class	437
Instantiating the STL String and Making Copies	437
Accessing Character Contents of a <code>std::string</code>	440
Concatenating One String to Another	442
Finding a Character or Substring in a String	444
Truncating an STL <code>string</code>	445
String Reversal	448
String Case Conversion	449
Template-Based Implementation of an STL String	450
C++14 operator <code>"s</code> in <code>std::string</code>	451
LESSON 17: STL Dynamic Array Classes	455
The Characteristics of <code>std::vector</code>	456
Typical Vector Operations	456
Instantiating a Vector	456
Inserting Elements at the End Using <code>push_back()</code>	458
List Initialization	459
Inserting Elements at a Given Position Using <code>insert()</code>	459
Accessing Elements in a Vector Using Array Semantics	462
Accessing Elements in a Vector Using Pointer Semantics	464
Removing Elements from a Vector	465
Understanding the Concepts of Size and Capacity	467
The STL <code>deque</code> Class	469
LESSON 18: STL <code>list</code> and <code>forward_list</code>	475
The Characteristics of a <code>std::list</code>	476
Basic <code>list</code> Operations	476
Instantiating a <code>std::list</code> Object	476
Inserting Elements at the Front or Back of the List	478

Inserting at the Middle of the List	479
Erasing Elements from the List	482
Reversing and Sorting Elements in a List	483
Reversing Elements Using <code>list::reverse()</code>	484
Sorting Elements	485
Sorting and Removing Elements from a <code>list</code> That Contains Instances of a <code>class</code>	487
<code>std::forward_list</code> Introduced in C++11	490
LESSON 19: STL Set Classes	495
An Introduction to STL Set Classes	496
Basic STL <code>set</code> and <code>multiset</code> Operations	496
Instantiating a <code>std::set</code> Object	497
Inserting Elements in a <code>set</code> or <code>multiset</code>	499
Finding Elements in an STL <code>set</code> or <code>multiset</code>	500
Erasing Elements in an STL <code>set</code> or <code>multiset</code>	502
Pros and Cons of Using STL <code>set</code> and <code>multiset</code>	507
STL Hash Set Implementation <code>std::unordered_set</code> and <code>std::unordered_multiset</code>	507
LESSON 20: STL Map Classes	513
An Introduction to STL Map Classes	514
Basic <code>std::map</code> and <code>std::multimap</code> Operations	515
Instantiating a <code>std::map</code> or <code>std::multimap</code>	515
Inserting Elements in an STL <code>map</code> or <code>multimap</code>	517
Finding Elements in an STL <code>map</code>	519
Finding Elements in an STL <code>multimap</code>	522
Erasing Elements from an STL <code>map</code> or <code>multimap</code>	522
Supplying a Custom Sort Predicate	525
STL's Hash Table-Based Key-Value Container	528
How Hash Tables Work	529
Using <code>unordered_map</code> and <code>unordered_multimap</code>	529
PART IV: More STL	
LESSON 21: Understanding Function Objects	537
The Concept of Function Objects and Predicates	538
Typical Applications of Function Objects	538

Unary Functions	538
Unary Predicate	543
Binary Functions	545
Binary Predicate	547
LESSON 22: Lambda Expressions	553
What Is a Lambda Expression?	554
How to Define a Lambda Expression	555
Lambda Expression for a Unary Function	555
Lambda Expression for a Unary Predicate	557
Lambda Expression with State via Capture Lists [...]	559
The Generic Syntax of Lambda Expressions	560
Lambda Expression for a Binary Function	562
Lambda Expression for a Binary Predicate	564
LESSON 23: STL Algorithms	569
What Are STL Algorithms?	570
Classification of STL Algorithms	570
Non-Mutating Algorithms	570
Mutating Algorithms	571
Usage of STL Algorithms	573
Finding Elements Given a Value or a Condition	573
Counting Elements Given a Value or a Condition	576
Searching for an Element or a Range in a Collection	577
Initializing Elements in a Container to a Specific Value	580
Using <code>std::generate()</code> to Initialize Elements to a Value Generated at Runtime	582
Processing Elements in a Range Using <code>for_each()</code>	583
Performing Transformations on a Range Using <code>std::transform()</code>	585
Copy and Remove Operations	588
Replacing Values and Replacing Element Given a Condition	590
Sorting and Searching in a Sorted Collection and Erasing Duplicates	592
Partitioning a Range	595
Inserting Elements in a Sorted Collection	597

LESSON 24: Adaptive Containers: Stack and Queue 603

The Behavioral Characteristics of Stacks and Queues	604
Stacks	604
Queues	604
Using the STL <code>stack</code> Class	605
Instantiating the Stack	605
Stack Member Functions	606
Insertion and Removal at Top Using <code>push()</code> and <code>pop()</code>	607
Using the STL <code>queue</code> Class	609
Instantiating the Queue	609
Member Functions of a <code>queue</code>	610
Insertion at End and Removal at the Beginning of <code>queue</code> via <code>push()</code> and <code>pop()</code>	611
Using the STL Priority Queue	613
Instantiating the <code>priority_queue</code> Class	613
Member Functions of <code>priority_queue</code>	615
Insertion at the End and Removal at the Beginning of <code>priority_queue</code> via <code>push()</code> and <code>pop()</code>	616

LESSON 25: Working with Bit Flags Using STL 621

The <code>bitset</code> Class	622
Instantiating the <code>std::bitset</code>	622
Using <code>std::bitset</code> and Its Members	623
Useful Operators Featured in <code>std::bitset</code>	624
<code>std::bitset</code> Member Methods	625
The <code>vector<bool></code>	627
Instantiating <code>vector<bool></code>	627
<code>vector<bool></code> Functions and Operators	628

PART V: Advanced C++ Concepts**LESSON 26: Understanding Smart Pointers 633**

What Are Smart Pointers?	634
The Problem with Using Conventional (Raw) Pointers	634
How Do Smart Pointers Help?	634
How Are Smart Pointers Implemented?	635
Types of Smart Pointers	636
Deep Copy	637
Copy on Write Mechanism	639

Reference-Counted Smart Pointers	639
Reference-Linked Smart Pointers	640
Destructive Copy	640
Using the <code>std::unique_ptr</code>	643
Popular Smart Pointer Libraries	645
LESSON 27: Using Streams for Input and Output	649
Concept of Streams	650
Important C++ Stream Classes and Objects	651
Using <code>std::cout</code> for Writing Formatted Data to Console	652
Changing Display Number Formats Using <code>std::cout</code>	653
Aligning Text and Setting Field Width Using <code>std::cout</code>	655
Using <code>std::cin</code> for Input	656
Using <code>std::cin</code> for Input into a Plain Old Data Type	656
Using <code>std::cin::get</code> for Input into <code>char*</code> Buffer	657
Using <code>std::cin</code> for Input into a <code>std::string</code>	658
Using <code>std::fstream</code> for File Handling	660
Opening and Closing a File Using <code>open()</code> and <code>close()</code>	660
Creating and Writing a Text File Using <code>open()</code> and <code>operator<<</code>	662
Reading a Text File Using <code>open()</code> and <code>operator>></code>	663
Writing to and Reading from a Binary File	664
Using <code>std::stringstream</code> for String Conversions	666
LESSON 28: Exception Handling	671
What Is an Exception?	672
What Causes Exceptions?	672
Implementing Exception Safety via <code>try</code> and <code>catch</code>	673
Using <code>catch(...)</code> to Handle All Exceptions	673
Catching Exception of a Type	674
Throwing Exception of a Type Using <code>throw</code>	676
How Exception Handling Works	677
Class <code>std::exception</code>	680
Your Custom Exception Class Derived from <code>std::exception</code>	680
LESSON 29: Going Forward	687
What's Different in Today's Processors?	688
How to Better Use Multiple Cores	689
What Is a Thread?	689
Why Program Multithreaded Applications?	690

How Can Threads Transact Data?	691
Using Mutexes and Semaphores to Synchronize Threads	692
Problems Caused by Multithreading	692
Writing Great C++ Code	693
C++17: Expected Features	694
if and switch Support Initializers	695
Copy Elision Guarantee	696
std::string_view Avoids Allocation Overheads	696
std::variant As a Typesafe Alternative to a union	697
Conditional Code Compilation Using if constexpr	697
Improved Lambda Expressions	698
Automatic Type Deduction for Constructors	698
template<auto>	699
Learning C++ Doesn't Stop Here!	699
Online Documentation	699
Communities for Guidance and Help	699
PART VI: Appendixes	
APPENDIX A: Working with Numbers: Binary and Hexadecimal	701
APPENDIX B: C++ Keywords	707
APPENDIX C: Operator Precedence	709
APPENDIX D: ASCII Codes	711
APPENDIX E: Answers	717
Index	763

Dedication

In memory of my father, who will continue to be my source of inspiration.

Acknowledgments

I am thankful to my family for the immense support, to my wife Clara, and to the editorial staff for their spirited engagement in getting this book to you!

About the Author

Siddhartha Rao is the Vice President in charge of Security Response at SAP SE, the world's leading supplier of enterprise software. The evolution of C++ convinces Siddhartha that you can program faster, simpler, and more powerful applications than ever before. He loves traveling and is a passionate mountain biker. He looks forward to your feedback on this effort!

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.

We welcome your comments. You can email or write 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 better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@sampublishing.com

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

Reader Services

Visit the publisher's website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that may be available for this book.

Introduction

2011 and 2014 were two special years for C++. While C++11 ushered in a dramatic improvement to C++, introducing new keywords and constructs that increased your programming efficiency, C++14 brought in incremental improvements that added finishing touches to the features introduced by C++11.

This book helps you learn C++ 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++ 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++ compiler for C++ and GNU's C++ compiler, which both offer a rich coverage of C++14 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. If you are a professional programmer, Part III, “Learning the Standard Template Library (STL),” is bound to help you create better, more practical C++ applications.

NOTE

Visit the publisher's website and register this book at **informit.com/register** for convenient access to any updates, downloads, or errata that may be available for this book.

Organization of This Book

Depending on your current proficiency levels with C++, you can choose the section you would like to start with. Concepts introduced by C++11 and C++14 are sprinkled throughout the book, in the relevant lessons. 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 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 to 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 keyword `auto` introduced by C++11 has made a significant reduction to the length of your iterator declarations. Lesson 22, “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++ applications, and introduces you to the new features expected to make it to the next version of the ISO standard called C++17.

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.

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 3

Using Variables, Declaring Constants

Variables are tools that help the programmer temporarily store data for a finite amount of time. *Constants* are tools that help the programmer define artifacts that are not allowed to change or make changes.

In this lesson, you find out

- How to declare and define variables and constants
- How to assign values to variables and manipulate those values
- How to write the value of a variable to the screen
- How to use keywords `auto` and `constexpr`

What Is a Variable?

Before you actually explore the need and use of variables in a programming language, take a step back and first see what a computer contains and how it works.

Memory and Addressing in Brief

All computers, smart phones, and other programmable devices contain a microprocessor and a certain amount of memory for temporary storage called Random Access Memory (RAM). In addition, many devices also allow for data to be persisted on a storage device such as the hard disk. The microprocessor executes your application, and in doing so it works with the RAM to fetch the application binary code to be executed as well as the data associated with it, which includes that displayed on the screen and that entered by the user.

The RAM itself can be considered to be a storage area akin to a row of lockers in the dorms, each locker having a number—that is, an address. To access a location in memory, say location 578, the processor needs to be asked via an instruction to fetch a value from there or write a value to it.

Declaring Variables to Access and Use Memory

The following examples will help you understand what variables are. Assume you are writing a program to multiply two numbers supplied by the user. The user is asked to feed the multiplicand and the multiplier into your program, one after the other, and you need to store each of them so that you can use them later to multiply. Depending on what you want to be doing with the result of the multiplication, you might even want to store it for later use in your program. It would be slow and error-prone if you were to explicitly specify memory addresses (such as 578) to store the numbers, as you would need to worry about inadvertently overwriting existing data at the location or your data being overwritten at a later stage.

When programming in languages like C++, you define variables to store those values. Defining a variable is quite simple and follows this pattern:

```
VariableType VariableName;
```

or

```
VariableType VariableName = InitialValue;
```

The variable type attribute tells the compiler the nature of data the variable can store, and the compiler reserves the necessary space for it. The name chosen by the programmer is a friendly replacement for the address in the memory where the variable's value is stored.

Unless the initial value is assigned, you cannot be sure of the contents of that memory location, which can be bad for the program. Therefore, initialization is optional, but it's often a good programming practice. Listing 3.1 shows how variables are declared, initialized, and used in a program that multiplies two numbers supplied by the user.

LISTING 3.1 Using Variables to Store Numbers and the Result of Their Multiplication

```
1: #include <iostream>
2: using namespace std;
3:
4: int main ()
5: {
6:     cout << "This program will help you multiply two numbers" << endl;
7:
8:     cout << "Enter the first number: ";
9:     int firstNumber = 0;
10:    cin >> firstNumber;
11:
12:    cout << "Enter the second number: ";
13:    int secondNumber = 0;
14:    cin >> secondNumber;
15:
16:    // Multiply two numbers, store result in a variable
17:    int multiplicationResult = firstNumber * secondNumber;
18:
19:    // Display result
20:    cout << firstNumber << " x " << secondNumber;
21:    cout << " = " << multiplicationResult << endl;
22:
23:    return 0;
24: }
```

Output ▼

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 24
51 x 24 = 1224
```

Analysis ▼

This application asks the user to enter two numbers, which the program multiplies and displays the result. To use numbers entered by the user, it needs to store them in the memory. Variables `firstNumber` and `secondNumber` declared in Lines 9 and 13 do the job of temporarily storing integer values entered by the user. You use `std::cin` in Lines 10 and 14 to accept input from the user and to store them in the two integer variables. The `cout` statement in Line 21 is used to display the result on the console.

Analyzing a variable declaration further:

```
9:    int firstNumber = 0;
```

What this line declares is a variable of type `int`, which indicates an integer, with a name called `firstNumber`. Zero is assigned to the variable as an initial value.

The compiler does the job of mapping this variable `firstNumber` to a location in memory and takes care of the associated memory-address bookkeeping for you for all the variables that you declare. The programmer thus works with human-friendly names, while the compiler manages memory-addressing and creates the instructions for the microprocessor to execute in working with the RAM.

CAUTION

Naming variables appropriately is important for writing good, understandable, and maintainable code.

Variable names in C++ can be alphanumeric, but they cannot start with a number. They cannot contain spaces and cannot contain arithmetic operators (such as `+`, `-`, and so on) within them. Variable names also cannot be reserved keywords. For example, a variable named `return` will cause compilation failure.

Variable names can contain the underscore character `_` that often is used in descriptive variable naming.

Declaring and Initializing Multiple Variables of a Type

In Listing 3.1, `firstNumber`, `secondNumber`, and `multiplicationResult` are all of the same type—integers—and are declared in three separate lines. If you wanted to, you could condense the declaration of these three variables to one line of code that looks like this:

```
int firstNumber = 0, secondNumber = 0, multiplicationResult = 0;
```

NOTE

As you can see, C++ makes it possible to declare multiple variables of a type at once and to declare variables at the beginning of a function. Yet, declaring a variable when it is first needed is often better as it makes the code readable—one notices the type of the variable when the declaration is close to its point of first use.

CAUTION

Data stored in variables is data stored in RAM. This data is lost when the application terminates unless the programmer explicitly persists the data on a storage medium like a hard disk.

Storing to a file on disk is discussed in Lesson 27, “Using Streams for Input and Output.”

Understanding the Scope of a Variable

Ordinary variables like the ones we have declared this far have a well-defined scope within which they’re valid and can be used. When used outside their scope, the variable names will not be recognized by the compiler and your program won’t compile. Beyond its scope, a variable is an unidentified entity that the compiler knows nothing of.

To better understand the scope of a variable, reorganize the program in Listing 3.1 into a function `MultiplyNumbers()` that multiplies the two numbers and returns the result. See Listing 3.2.

3

LISTING 3.2 Demonstrating the Scope of the Variables

```
1: #include <iostream>
2: using namespace std;
3:
4: void MultiplyNumbers ()
5: {
6:     cout << "Enter the first number: ";
7:     int firstNumber = 0;
8:     cin >> firstNumber;
9:
10:    cout << "Enter the second number: ";
11:    int secondNumber = 0;
12:    cin >> secondNumber;
13:
14:    // Multiply two numbers, store result in a variable
15:    int multiplicationResult = firstNumber * secondNumber;
16:
17:    // Display result
18:    cout << firstNumber << " x " << secondNumber;
19:    cout << " = " << multiplicationResult << endl;
20: }
21: int main ()
22: {
23:     cout << "This program will help you multiply two numbers" << endl;
24:
25:     // Call the function that does all the work
26:     MultiplyNumbers();
```

```
27:
28:     // cout << firstNumber << " x " << secondNumber;
29:     // cout << " = " << multiplicationResult << endl;
30:
31:     return 0;
32: }
```

Output ▼

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 24
51 x 24 = 1224
```

Analysis ▼

Listing 3.2 does exactly the same activity as Listing 3.1 and produces the same output. The only difference is that the bulk of the work is delegated to a function called `MultiplyNumbers()` invoked by `main()`. Note that variables `firstNumber` and `secondNumber` cannot be used outside of `MultiplyNumbers()`. If you uncomment Lines 28 or 29 in `main()`, you experience compile failure of type undeclared identifier.

This is because the scope of the variables `firstNumber` and `secondNumber` is local, hence limited to the function they're declared in, in this case `MultiplyNumbers()`. A local variable can be used in a function after variable declaration till the end of the function. The curly brace (`}`) that indicates the end of a function also limits the scope of variables declared in the same. When a function ends, all local variables are destroyed and the memory they occupied returned.

When compiled, variables declared within `MultiplyNumbers()` perish when the function ends, and if they're used in `main()`, compilation fails as the variables have not been declared in there.

CAUTION

If you declare another set of variables with the same name in `main()`, then don't still expect them to carry a value that might have been assigned in `MultiplyNumbers()`.

The compiler treats the variables in `main()` as independent entities even if they share their names with a variable declared in another function, as the two variables in question are limited by their scope.

Global Variables

If the variables used in function `MultiplyNumbers()` in Listing 3.2 were declared outside the scope of the function `MultiplyNumber()` instead of within it, then they would be usable in both `main()` and `MultiplyNumbers()`. Listing 3.3 demonstrates global variables, which are the variables with the widest scope in a program.

LISTING 3.3 Using Global Variables

```
1: #include <iostream>
2: using namespace std;
3:
4: // three global integers
5: int firstNumber = 0;
6: int secondNumber = 0;
7: int multiplicationResult = 0;
8:
9: void MultiplyNumbers ()
10: {
11:     cout << "Enter the first number: ";
12:     cin >> firstNumber;
13:
14:     cout << "Enter the second number: ";
15:     cin >> secondNumber;
16:
17:     // Multiply two numbers, store result in a variable
18:     multiplicationResult = firstNumber * secondNumber;
19:
20:     // Display result
21:     cout << "Displaying from MultiplyNumbers(): ";
22:     cout << firstNumber << " x " << secondNumber;
23:     cout << " = " << multiplicationResult << endl;
24: }
25: int main ()
26: {
27:     cout << "This program will help you multiply two numbers" << endl;
28:
29:     // Call the function that does all the work
30:     MultiplyNumbers();
31:
32:     cout << "Displaying from main(): ";
33:
34:     // This line will now compile and work!
35:     cout << firstNumber << " x " << secondNumber;
36:     cout << " = " << multiplicationResult << endl;
37:
38:     return 0;
39: }
```

Output ▼

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 19
Displaying from MultiplyNumbers(): 51 x 19 = 969
Displaying from main(): 51 x 19 = 969
```

Analysis ▼

Listing 3.3 displays the result of multiplication in two functions, neither of which has declared the variables `firstNumber`, `secondNumber`, and `multiplicationResult`. These variables are `global` as they have been declared in Lines 5–7, outside the scope of any function. Note Lines 23 and 36 that use these variables and display their values. Pay special attention to how `multiplicationResult` is first assigned in `MultiplyNumbers()` yet is effectively reused in `main()`.

CAUTION

Indiscriminate use of global variables is considered poor programming practice. This is because global variables can be assigned values in any/every function and can contain an unpredictable state, especially when functions that modify them run in different threads or are programmed by different programmers in a team.

An elegant way of programming Listing 3.3 without using global variables would have the function `MultiplyNumbers()` return the integer result of the multiplication to `main()`.

Naming Conventions

In case you haven't noticed, we named the function `MultiplyNumbers()` where every word in the function name starts with a capital letter (called *Pascal casing*), while variables `firstNumber`, `secondNumber`, and `multiplicationResult` were given names where the first word starts with a lowercase letter (called *camel casing*). This book follows a convention where variable names follow camel casing, while other artifacts such as function names follow Pascal casing.

You may come across C++ code wherein a variable name is prefixed with characters that explain the type of the variable. This convention is called the *Hungarian notation*.

and is frequently used in the programming of Windows applications. So, `firstNumber` in Hungarian notation would be `iFirstNumber`, where the prefix `i` stands for integer. A global integer would be called `g_iFirstNumber`. Hungarian notation has lost popularity in recent years in part due to improvements in Integrated Development Environments (IDEs) that display the type of a variable when required—on mouse hover, for instance.

Examples of commonly found bad variable names follow:

```
int i = 0;
bool b = false;
```

The name of the variable should indicate its purpose, and the two can be better declared as

```
int totalCash = 0;
bool isLampOn = false;
```

CAUTION

Naming conventions are used to make the code readable to programmers, not to compilers. So choose a convention that suits wisely and use it consistently.

When working in a team, it is a good idea to align on the convention to be used before starting a new project. When working on an existing project, adopt the used convention so that the new code remains readable to others.

Common Compiler-Supported C++ Variable Types

In most of the examples thus far, you have defined variables of type `int`—that is, integers. However, C++ programmers can choose from a variety of fundamental variable types supported directly by the compiler. Choosing the right variable type is as important as choosing the right tools for the job! A Phillips screwdriver won't work well with a regular screw head just like an unsigned integer can't be used to store values that are negative! Table 3.1 enlists the various variable types and the nature of data they can contain.

TABLE 3.1 Variable Types

Type	Values
bool	true or false
char	256 character values
unsigned short int	0 to 65,535
short int	−32,768 to 32,767
unsigned long int	0 to 4,294,967,295
long int	−2,147,483,648 to 2,147,483,647
unsigned long long	0 to 18,446,744,073,709,551,615
long long	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int (16 bit)	−32,768 to 32,767
int (32 bit)	−2,147,483,648 to 2,147,483,647
unsigned int (16 bit)	0 to 65,535
unsigned int (32 bit)	0 to 4,294,967,295
float	1.2e−38 to 3.4e38
double	2.2e−308 to 1.8e308

The following sections explain the important types in greater detail.

Using Type `bool` to Store Boolean Values

C++ provides a type that is specially created for containing Boolean values `true` or `false`, both of which are reserved C++ keywords. This type is particularly useful in storing settings and flags that can be ON or OFF, present or absent, available or unavailable, and the like.

A sample declaration of an initialized Boolean variable is

```
bool alwaysOnTop = false;
```

An expression that evaluates to a Boolean type is

```
bool deleteFile = (userSelection == "yes");  
// evaluates to true if userSelection contains "yes", else to false
```

Conditional expressions are explained in Lesson 5, “Working with Expressions, Statements, and Operators.”

Using Type `char` to Store Character Values

Use type `char` to store a single character. A sample declaration is

```
char userInput = 'Y'; // initialized char to 'Y'
```

Note that memory is comprised of bits and bytes. Bits can be either 0 or 1, and bytes can contain numeric representation using these bits. So, working or assigning character data as shown in the example, the compiler converts the character into a numeric representation that can be placed into memory. The numeric representation of Latin characters A–Z, a–z, numbers 0–9, some special keystrokes (for example, DEL), and special characters (such as backspace) has been standardized by the American Standard Code for Information Interchange, also called ASCII.

You can look up the table in Appendix D, “ASCII Codes,” to see that the character Y assigned to variable `userInput` has the ASCII value 89 in decimal. Thus, what the compiler does is store 89 in the memory space allocated for `userInput`.

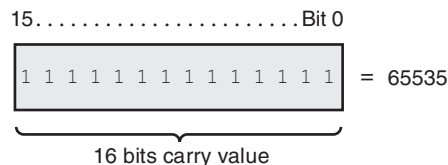
The Concept of Signed and Unsigned Integers

Sign implies positive or negative. All numbers you work with using a computer are stored in the memory in the form of bits and bytes. A memory location that is 1 byte large contains 8 bits. Each bit can either be a 0 or 1 (that is, carry one of these two values at best). Thus, a memory location that is 1 byte large can contain a maximum of 2 to the power 8 values—that is, 256 unique values. Similarly, a memory location that is 16 bits large can contain 2 to the power 16 values—that is, 65,536 unique values.

If these values were to be unsigned—assumed to be only positive—then one byte could contain integer values ranging from 0 through 255 and two bytes would contain values ranging from 0 through 65,535, respectively. Look at Table 3.1 and note that the unsigned `short` is the type that supports this range, as it is contained in 16 bits of memory. Thus, it is quite easy to model positive values in bits and bytes (see Figure 3.1).

FIGURE 3.1

Organization of bits in a 16-bit unsigned short integer.

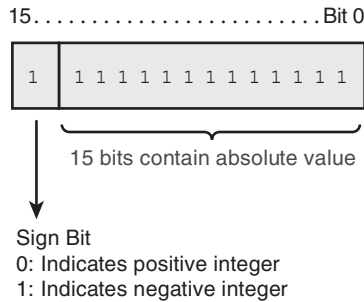


How to model negative numbers in this space? One way is to “sacrifice” a bit as the sign-bit that would indicate if the values contained in the other bits are positive or

negative (see Figure 3.2). The sign-bit needs to be the most-significant-bit (MSB) as the least-significant-one would be required to model odd numbers. So, when the MSB contains sign-information, it is assumed that 0 would be positive and 1 would mean negative, and the other bytes contain the absolute value.

FIGURE 3.2

Organization of bits in a 16-bit signed short integer.



Thus, a signed number that occupies 8 bits can contain values ranging from -128 through 127 , and one that occupies 16 bits can contain values ranging from $-32,768$ through $32,767$. If you look at Table 3.1 again, note that the (signed) `short` is the type that supports positive and negative integer values in a 16-bit space.

Signed Integer Types `short`, `int`, `long`, and `long long`

These types differ in their sizes and thereby differ in the range of values they can contain. `int` is possibly the most used type and is 32 bits wide on most compilers. Use the right type depending on your projection of the maximum value that particular variable would be expected to hold.

Declaring a variable of a signed type is simple:

```
short int gradesInMath = -5; // not your best score
int moneyInBank = -70000; // overdraft
long populationChange = -85000; // reducing population
long long countryGDPChange = -70000000000;
```

Unsigned Integer Types `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`

Unlike their signed counterparts, unsigned integer variable types cannot contain sign information, and hence they can actually support twice as many positive values.

Declaring a variable of an unsigned type is as simple as this:

```
unsigned short int numColorsInRainbow = 7;  
unsigned int numEggsInBasket = 24; // will always be positive  
unsigned long numCarsInNewYork = 700000;  
unsigned long long countryMedicareExpense = 70000000000;
```

NOTE

You would use an unsigned variable type when you expect only positive values. So, if you're counting the number of apples, don't use `int`; use `unsigned int`. The latter can hold twice as many values in the positive range as the former can.

CAUTION

So, an unsigned type might not be suited for a variable in a banking application used to store the account balance as banks do allow some customers an overdraft facility. To see an example that demonstrates the differences between signed and unsigned types, visit Listing 5.3 in Lesson 5.

3

Avoid Overflow Errors by Selecting Correct Data Types

Data types such as `short`, `int`, `long`, `unsigned short`, `unsigned int`, `unsigned long`, and the like have a finite capacity for containing numbers. When you exceed the limit imposed by the type chosen in an arithmetic operation, you create an overflow.

Take `unsigned short` for an example. Data type `short` consumes 16 bits and can hence contain values from 0 through 65,535. When you add 1 to 65,535 in an `unsigned short`, the value overflows to 0. It's like the odometer of a car that suffers a mechanical overflow when it can support only five digits and the car has done 99,999 kilometers (or miles).

In this case, `unsigned short` was never the right type for such a counter. The programmer was better off using `unsigned int` to support numbers higher than 65,535.

In the case of a signed `short` integer, which has a range of $-32,768$ through $32,767$, adding 1 to 32,767 may result in the signed integer taking the highest negative value. This behavior is compiler dependent.

Listing 3.4 demonstrates the overflow errors that you can inadvertently introduce via arithmetic operations.

LISTING 3.4 Demonstrating the Ill-Effects of Signed and Unsigned Integer Overflow Errors

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     unsigned short uShortValue = 65535;
6:     cout << "Incrementing unsigned short " << uShortValue << " gives: ";
7:     cout << ++uShortValue << endl;
8:
9:     short signedShort = 32767;
10:    cout << "Incrementing signed short " << signedShort << " gives: ";
11:    cout << ++signedShort << endl;
12:
13:    return 0;
14: }
```

Output ▼

```
Incrementing unsigned short 65535 gives: 0
Incrementing signed short 32767 gives: -32768
```

Analysis ▼

The output indicates that unintentional overflow situations result in unpredictable and unintuitive behavior for the application. Lines 7 and 11 increment an `unsigned short` and a `signed short` that have previously been initialized to their maximum supported values—65,535 and 32,767, respectively. The output demonstrates the value they hold after the increment operation, namely an overflow of 65,535 to zero in the unsigned short and an overflow of 32,767 to -32,768 in the signed short. One wouldn't expect the result of an increment operation to reduce the value in question, but that is exactly what happens when an integer type overflows. If you were using the values in question to allocate memory, then with the `unsigned short`, you can reach a point where you request zero bytes when your actual need is 65,536 bytes.

NOTE

The operations `++uShortValue` and `++signedShort` seen in Listing 3.4 at lines 7 and 11 are prefix increment operations. These are explained in detail in Lesson 5.

Floating-Point Types `float` and `double`

Floating-point numbers are what you might have learned in school as real numbers. These are numbers that can be positive or negative. They can contain decimal values. So, if you want to store the value of pi ($22 / 7$ or 3.14) in a variable in C++, you would use a floating-point type.

Declaring variables of these types follows exactly the same pattern as the `int` in Listing 3.1. So, a `float` that allows you to store decimal values would be declared as the following:

```
float pi = 3.14;
```

And a double precision `float` (called simply a `double`) is defined as

```
double morePrecisePi = 22.0 / 7;
```

TIP

C++14 adds support for *chunking separators* in the form of a single quotation mark. This improves readability of code, as seen in the following initializations:

```
int moneyInBank = -70'000; // -70000
long populationChange = -85'000; // -85000
long long countryGDPChange = -70'000'000'000; //
-70 billion
double pi = 3.141'592'653'59; // 3.14159265359
```

NOTE

The data types mentioned thus far are often referred to as POD (Plain Old Data). The category POD contains these as well as aggregations (structs, enums, unions, or classes) thereof.

Determining the Size of a Variable

Using `sizeof`

Size is the amount of memory that the compiler reserves when the programmer declares a variable to hold the data assigned to it. The size of a variable depends on its type, and C++ has a very convenient operator called `sizeof` that tells you the size in bytes of a variable or a type.

The usage of `sizeof` is simple. To determine the size of an integer, you invoke `sizeof` with parameter `int` (the type) as demonstrated by Listing 3.5.

```
cout << "Size of an int: " << sizeof (int);
```

LISTING 3.5 Finding the Size of Standard C++ Variable Types

```
1: #include <iostream>
2:
3: int main()
4: {
5:     using namespace std;
6:     cout << "Computing the size of some C++ inbuilt variable types" << endl;
7:
8:     cout << "Size of bool: " << sizeof(bool) << endl;
9:     cout << "Size of char: " << sizeof(char) << endl;
10:    cout << "Size of unsigned short int: " << sizeof(unsigned short) << endl;
11:    cout << "Size of short int: " << sizeof(short) << endl;
12:    cout << "Size of unsigned long int: " << sizeof(unsigned long) << endl;
13:    cout << "Size of long: " << sizeof(long) << endl;
14:    cout << "Size of int: " << sizeof(int) << endl;
15:    cout << "Size of unsigned long long: " << sizeof(unsigned long long) <<
endl;
16:    cout << "Size of long long: " << sizeof(long long) << endl;
17:    cout << "Size of unsigned int: " << sizeof(unsigned int) << endl;
18:    cout << "Size of float: " << sizeof(float) << endl;
19:    cout << "Size of double: " << sizeof(double) << endl;
20:
21:    cout << "The output changes with compiler, hardware and OS" << endl;
22:
23:    return 0;
24: }
```

Output ▼

Computing the size of some C++ inbuilt variable types

Size of `bool`: 1

Size of `char`: 1

Size of `unsigned short int`: 2

Size of `short int`: 2

Size of `unsigned long int`: 4

Size of `long`: 4

Size of `int`: 4

Size of `unsigned long long`: 8

Size of `long long`: 8

Size of `unsigned int`: 4

Size of `float`: 4

Size of `double`: 8

The output changes with compiler, hardware and OS

Analysis ▼

The output of Listing 3.5 reveals sizes of various types in bytes and is specific to my platform: compiler, OS, and hardware. This output in particular is a result of running the program in 32-bit mode (compiled by a 32-bit compiler) on a 64-bit operating system. Note that a 64-bit compiler probably creates different results, and the reason I chose a 32-bit compiler was to be able to run the application on 32-bit as well as 64-bit systems. The output tells that the `sizeof` of a variable doesn't change between an unsigned or signed type; the only difference in the two is the MSB that carries sign information in the former.

NOTE

All sizes seen in the output are in bytes. The size of a type is an important parameter to be considered, especially for types used to hold numbers. A `short int` can hold a smaller range than a `long long`. You therefore wouldn't use a `short int` to hold the population of a country, for example.

TIP

C++11 introduced fixed-width integer types that allow you to specify the exact width of the integer in bits. These are `int8_t` or `uint8_t` for 8-bit signed and unsigned integers, respectively. You may also use 16-bit (`int16_t`, `uint16_t`), 32-bit (`int32_t`, `uint32_t`), and 64-bit (`int64_t`, `uint64_t`) integer types. To use these types, remember to include header `<cstdint>`.

Avoid Narrowing Conversion Errors by Using List Initialization

When you initialize a variable of a smaller integer type (say, `short`) using another of a larger type (say, an `int`), you are risking a narrowing conversion error, because the compiler has to fit data stored in a type that can potentially hold much larger numbers into a type that doesn't have the same capacity (that is, is narrower). Here's an example:

```
int largeNum = 5000000;
short smallNum = largeNum; // compiles OK, yet narrowing error
```

Narrowing isn't restricted to conversions between integer types only. You may face narrowing errors if you initialize a `float` using a `double`, a `float` (or `double`) using an `int`, or an `int` using a `float`. Some compilers may warn, but this warning will not cause an error that stops compilation. In such cases, you may be confronted by bugs that occur infrequently and at execution time.

To avoid this problem, C++11 recommends *list initialization* techniques that prevent narrowing. To use this feature, insert initialization values/variables within braces {...}. The list initialization syntax is as follows:

```
int largeNum = 5000000;
short anotherNum{ largeNum }; // error! Amend types
int anotherNum{ largeNum }; // OK!
float someFloat{ largeNum }; // error! An int may be narrowed
float someFloat{ 5000000 }; // OK! 5000000 can be accommodated
```

It may not be immediately apparent, but this feature has the potential to spare bugs that occur when data stored in a type undergoes a narrowing conversion at execution time—these occur implicitly during an initialization and are tough to solve.

Automatic Type Inference Using `auto`

There are cases where the type of a variable is apparent given the initialization value being assigned to it. For example, if a variable is being initialized with the value `true`, the type of the variable can be best estimated as `bool`. Compilers supporting C++11 and beyond give you the option of not having to explicitly specify the variable type when using the keyword `auto`.

```
auto coinFlippedHeads = true;
```

We have left the task of defining an exact type for variable `coinFlippedHeads` to the compiler. The compiler checks the nature of the value the variable is being initialized to and then decides on the best possible type that suits this variable. In this particular case, it is clear that an initialization value of `true` best suits a variable that is of type `bool`. The compiler thus determines `bool` as the type that suits variable `coinFlippedHeads` best and internally treats `coinFlippedHeads` as a `bool`, as also demonstrated by Listing 3.6.

LISTING 3.6 Using the `auto` Keyword and Relying on the Compiler's Type-Inference Capabilities

```
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     auto coinFlippedHeads = true;
7:     auto largeNumber = 2500000000000;
8:
9:     cout << "coinFlippedHeads = " << coinFlippedHeads;
10:    cout << " , sizeof(coinFlippedHeads) = " << sizeof(coinFlippedHeads) <<
endl;
11:    cout << "largeNumber = " << largeNumber;
12:    cout << " , sizeof(largeNumber) = " << sizeof(largeNumber) << endl;
13:
14:    return 0;
15: }
```

Output ▼

```
coinFlippedHeads = 1 , sizeof(coinFlippedHeads) = 1
largeNumber = 2500000000000 , sizeof(largeNumber) = 8
```

Analysis ▼

See how instead of deciding that `coinFlippedHeads` should be of type `bool` or that `largeNumber` should be a `long long`, you have used the `auto` keyword in Lines 6 and 7 where the two variables have been declared. This delegates the decision on the type of variable to the compiler, which uses the initialization value as a ballpark. You have used `sizeof` to actually check whether the compiler created the types you suspected it would, and you can check against the output produced by your code to verify that it really did.

NOTE

Using `auto` requires you to initialize the variable for the compiler that uses this initial value in deciding what the variable type can be.

When you don't initialize a variable of type `auto`, you get a compile error.

Even if `auto` seems to be a trivial feature at first sight, it makes programming a lot easier in those cases where the type variable is a complex type. The role of `auto` in writing simpler, yet type-safe code is revisited in Lesson 15, “An Introduction to the Standard Template Library,” and beyond.

Using `typedef` to Substitute a Variable's Type

C++ allows you to substitute variable types to something that you might find convenient. You use the keyword `typedef` for that. Here is an example where a programmer wants to call an `unsigned int` a descriptive `STRICTLY_POSITIVE_INTEGER`.

```
typedef unsigned int STRICTLY_POSITIVE_INTEGER;  
STRICTLY_POSITIVE_INTEGER numEggsInBasket = 4532;
```

When compiled, the first line tells the compiler that a `STRICTLY_POSITIVE_INTEGER` is nothing but an `unsigned int`. At later stages when the compiler encounters the already defined type `STRICTLY_POSITIVE_INTEGER`, it substitutes it for `unsigned int` and continues compilation.

NOTE

`typedef` or type substitution is particularly convenient when dealing with complex types that can have a cumbersome syntax, for example, types that use templates. Templates are discussed later in Lesson 14, “An Introduction to Macros and Templates.”

What Is a Constant?

Imagine you are writing a program to calculate the area and the circumference of a circle. The formulas are

```
Area = pi * Radius * Radius;  
Circumference = 2 * pi * Radius
```

In this formula, `pi` is the constant of value `22 / 7`. You don't want the value of `pi` to change anywhere in your program. You also want to avoid any accidental assignments of possibly incorrect values to `pi`. C++ enables you to define `pi` as a constant that cannot be changed after declaration. In other words, after it's defined, the value of a constant cannot be altered. Assignments to a constant in C++ cause compilation errors.

Thus, constants are like variables in C++ except that these cannot be changed. Similar to variables, constants also occupy space in the memory and have a name to identify the address where the space is reserved. However, the content of this space cannot be overwritten. Constants in C++ can be

- Literal constants
- Declared constants using the `const` keyword
- Constant expressions using the `constexpr` keyword (new since C++11)
- Enumerated constants using the `enum` keyword
- Defined constants that are not recommended and deprecated

Literal Constants

Literal constants can be of many types—integer, string, and so on. In your first C++ program in Listing 1.1, you displayed “Hello World” using the following statement:

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

In here, “Hello World” is a string literal constant. You literally have been using literal constants all the while! When you declare an integer `someNumber`, like this:

```
int someNumber = 10;
```

The integer variable `someNumber` is assigned an initial value of ten. Here decimal ten is a part of the code, gets compiled into the application, is unchangeable, and is a literal constant too. You may initialize the integer using a literal in octal notation, like this:

```
int someNumber = 012 // octal 12 evaluates to decimal 10
```

Starting in C++14, you may also use binary literals, like this:

```
int someNumber = 0b1010; // binary 1010 evaluates to decimal 10
```

TIP

C++ also allows you to define your own literals. For example, temperature as `32.0_F` (Fahrenheit) or `0.0_C` (Centigrade), distance as `16_m` (Miles) or `10_km` (Kilometers), and so on.

These suffixes `_F`, `_C`, `_m`, and `_km` are called user-defined literals and are explained in Lesson 12, “Operator Types and Operator Overloading,” after the prerequisite concepts are explained.

Declaring Variables as Constants Using `const`

The most important type of constants in C++ from a practical and programmatic point of view are declared by using keyword `const` before the variable type. The generic declaration looks like the following:

```
const type-name constant-name = value;
```

Let’s see a simple application that displays the value of a constant called `pi` (see Listing 3.7).

LISTING 3.7 Declaring a Constant Called `pi`

```
1: #include <iostream>
2:
3: int main()
4: {
5:     using namespace std;
6:
7:     const double pi = 22.0 / 7;
8:     cout << "The value of constant pi is: " << pi << endl;
9:
10:    // Uncomment next line to view compile failure
11:    // pi = 345;
12:
13:    return 0;
14: }
```

Output ▼

The value of constant pi is: 3.14286

Analysis ▼

Note the declaration of constant `pi` in Line 7. We use the `const` keyword to tell the compiler that `pi` is a constant of type `double`. If you uncomment Line 11 where the

programmer tries to assign a value to a variable you have defined as a constant, you see a compile failure that says something similar to, “You cannot assign to a variable that is `const`.” Thus, constants are a powerful way to ensure that certain data cannot be modified.

NOTE

It is good programming practice to define variables that are not supposed to change their values as `const`. The usage of the `const` keyword indicates that the programmer has thought about ensuring the constantness of data where required and protects his application from inadvertent changes to this constant.

This is particularly useful in a multiprogrammer environment.

Constants are useful when declaring the length of static arrays, which are fixed at compile time. Listing 4.2 in Lesson 4, “Managing Arrays and Strings,” includes a sample that demonstrates the use of a `const int` to define the length of an array.

Constant Expressions Using `constexpr`

Keyword `constexpr` allows function-like declaration of constants:

```
constexpr double GetPi() {return 22.0 / 7;}
```

One `constexpr` can use another:

```
constexpr double TwicePi() {return 2 * GetPi();}
```

`constexpr` may look like a function, however, allows for optimization possibilities from the compiler’s and application’s point of view. So long as a compiler is capable of evaluating a constant expression to a constant, it can be used in statements and expressions at places where a constant is expected. In the preceding example, `TwicePi()` is a `constexpr` that uses a constant expression `GetPi()`. This will possibly trigger a compile-time optimization wherein every usage of `TwicePi()` is simply replaced by `6.28571` by the compiler, and not the code that would calculate `2 x 22 / 7` when executed.

Listing 3.8 demonstrates the usage of `constexpr`.

LISTING 3.8 Using `constexpr` to Calculate Pi

```
1: #include <iostream>
2: constexpr double GetPi() { return 22.0 / 7; }
3: constexpr double TwicePi() { return 2 * GetPi(); }
4:
5: int main()
```

```
6: {  
7:     using namespace std;  
8:     const double pi = 22.0 / 7;  
9:  
10:    cout << "constant pi contains value " << pi << endl;  
11:    cout << "constexpr GetPi() returns value " << GetPi() << endl;  
12:    cout << "constexpr TwicePi() returns value " << TwicePi() << endl;  
  
13:    return 0;  
14: }
```

Output ▼

```
constant pi contains value 3.14286  
constexpr GetPi() returns value 3.14286  
constexpr TwicePi() returns value 6.28571
```

Analysis ▼

The program demonstrates two methods of deriving the value of `pi`—one as a constant variable `pi` as declared in Line 8 and another as a constant expression `GetPi()` declared in Line 2. `GetPi()` and `TwicePi()` may look like functions, but they are not exactly. Functions are invoked at program execution time. But, these are constant expressions and the compiler had already substituted every usage of `GetPi()` by 3.14286 and every usage of `TwicePi()` by 6.28571. Compile-time resolution of `TwicePi()` increases the speed of program execution when compared to the same calculation being contained in a function.

NOTE

Constant expressions need to contain simple implementations that return simple types like integer, double, and so on. C++14 allows `constexpr` to contain decision-making constructs such as `if` and `switch` statements. These conditional statements are discussed in detail in Lesson 6, “Controlling Program Flow.”

The usage of `constexpr` will not guarantee compile-time optimization—for example, if you use a `constexpr` expression to double a user provided number. The outcome of such an expression cannot be calculated by the compiler, which may ignore the usage of `constexpr` and compile as a regular function.

To see a demonstration of how a constant expression is used in places where the compiler expects a constant, see the code sample in Listing 4.2 in Lesson 4.

TIP

In the previous code samples, we defined our own constant `pi` as an exercise in learning the syntax of declaring constants and `constexpr`. Yet, most popular C++ compilers already supply you with a reasonably precise value of `pi` in the constant `M_PI`. You may use this constant in your programs after including header file `<cmath>`.

Enumerations

There are situations where a particular variable should be allowed to accept only a certain set of values. These are situations where you don't want the colors in the rainbow to contain Turquoise or the directions on a compass to contain Left. In both these cases, you need a type of variable whose values are restricted to a certain set defined by you. *Enumerations* are exactly the tool you need in this situation and are characterized by the keyword `enum`. Enumerations comprise a set of constants called *enumerators*.

In the following example, the enumeration `RainbowColors` contains individual colors such as `Violet` as enumerators:

```
enum RainbowColors
{
    Violet = 0,
    Indigo,
    Blue,
    Green,
    Yellow,
    Orange,
    Red
};
```

Here's another enumeration for the cardinal directions:

```
enum CardinalDirections
{
    North,
    South,
    East,
    West
};
```

Enumerations are used as user-defined types. Variables of this type can be assigned a range of values restricted to the enumerators contained in the enumeration. So, if defining a variable that contains the colors of a rainbow, you declare the variable like this:

```
RainbowColors MyFavoriteColor = Blue; // Initial value
```


In the preceding line of code, you declared an enumerated constant `MyFavoriteColor` of type `RainbowColors`. This enumerated constant variable is restricted to contain any of the legal `VIBGYOR` colors and no other value.

NOTE

The compiler converts the enumerator such as `Violet` and so on into integers. Each enumerated value specified is one more than the previous value. You have the choice of specifying a starting value, and if this is not specified, the compiler takes it as 0. So, `North` is evaluated as value 0.

If you want, you can also specify an explicit value against each of the enumerated constants by initializing them.

Listing 3.9 demonstrates how enumerated constants are used to hold the four cardinal directions, with an initializing value supplied to the first one.

LISTING 3.9 Using Enumerated Values to Indicate Cardinal Wind Directions

```
1: #include <iostream>
2: using namespace std;
3:
4: enum CardinalDirections
5: {
6:     North = 25,
7:     South,
8:     East,
9:     West
10: };
11:
12: int main()
13: {
14:     cout << "Displaying directions and their symbolic values" << endl;
15:     cout << "North: " << North << endl;
16:     cout << "South: " << South << endl;
17:     cout << "East: " << East << endl;
18:     cout << "West: " << West << endl;
19:
20:     CardinalDirections windDirection = South;
21:     cout << "Variable windDirection = " << windDirection << endl;
22:
23:     return 0;
24: }
```

Output ▼

```
Displaying directions and their symbolic values
North: 25
South: 26
East: 27
West: 28
Variable windDirection = 26
```

Analysis ▼

Note how we have enumerated the four cardinal directions but have given the first `North` an initial value of 25 (see Line 6). This automatically ensures that the following constants are assigned values 26, 27, and 28 by the compiler as demonstrated in the output. In Line 20 you create a variable of type `CardinalDirections` that is assigned an initial value `South`. When displayed on the screen in Line 21, the compiler dispatches the integer value associated with `South`, which is 26.

TIP

You may want to take a look at Listings 6.4 and 6.5 in Lesson 6. They use `enum` to enumerate the days of the week and conditional processing to tell what the day of the user's choosing is named after.

Defining Constants Using `#define`

First and foremost, don't use this if you are writing a program anew. The only reason this book analyzes the definition of constants using `#define` is to help you understand certain legacy programs that do define constants such as `pi` using this syntax:

```
#define pi 3.14286
```

`#define` is a preprocessor macro, and what is done here is that all mentions of `pi` henceforth are replaced by 3.14286 for the compiler to process. Note that this is a text replacement (read: non-intelligent replacement) done by the preprocessor. The compiler neither knows nor cares about the actual type of the constant in question.

CAUTION

Defining constants using the preprocessor via `#define` is deprecated and should not be used.

Keywords You Cannot Use as Variable or Constant Names

Some words are reserved by C++, and you cannot use them as variable names. These keywords have special meaning to the C++ compiler. Keywords include `if`, `while`, `for`, and `main`. A list of keywords defined by C++ is presented in Table 3.2 as well as in Appendix B, “C++ Keywords.” Your compiler might have additional reserved words, so you should check its manual for a complete list.

TABLE 3.2 Major C++ Keywords

<code>asm</code>	<code>else</code>	<code>new</code>	<code>this</code>
<code>auto</code>	<code>enum</code>	<code>operator</code>	<code>throw</code>
<code>bool</code>	<code>explicit</code>	<code>private</code>	<code>true</code>
<code>break</code>	<code>export</code>	<code>protected</code>	<code>try</code>
<code>case</code>	<code>extern</code>	<code>public</code>	<code>typedef</code>
<code>catch</code>	<code>false</code>	<code>register</code>	<code>typeid</code>
<code>char</code>	<code>float</code>	<code>reinterpret_cast</code>	<code>typename</code>
<code>class</code>	<code>for</code>	<code>return</code>	<code>union</code>
<code>const</code>	<code>friend</code>	<code>short</code>	<code>unsigned</code>
<code>constexpr</code>	<code>goto</code>	<code>signed</code>	<code>using</code>
<code>continue</code>	<code>if</code>	<code>sizeof</code>	<code>virtual</code>
<code>default</code>	<code>inline</code>	<code>static</code>	<code>void</code>
<code>delete</code>	<code>int</code>	<code>static_cast</code>	<code>volatile</code>
<code>do</code>	<code>long</code>	<code>struct</code>	<code>wchar_t</code>
<code>double</code>	<code>mutable</code>	<code>switch</code>	<code>while</code>
<code>dynamic_cast</code>	<code>namespace</code>	<code>template</code>	
In addition, the following words are reserved:			
<code>and</code>	<code>bitor</code>	<code>not_eq</code>	<code>xor</code>
<code>and_eq</code>	<code>compl</code>	<code>or</code>	<code>xor_eq</code>
<code>bitand</code>	<code>not</code>	<code>or_eq</code>	

DO	DON'T
<p>DO give variables descriptive names, even if that makes them long.</p> <p>DO initialize variables, and use list initialization to avoid narrowing conversion errors.</p> <p>DO ensure that the name of the variable explains its purpose.</p> <p>DO put yourself into the shoes of one who hasn't seen your code yet and think whether the name would make sense to him or her.</p> <p>DO check whether your team is following certain naming conventions and follow them.</p>	<p>DON'T give names that are too short or contain just a character.</p> <p>DON'T give names that use exotic acronyms known only to you.</p> <p>DON'T give names that are reserved C++ keywords as these won't compile.</p>

Summary

In this lesson you learned about using memory to store values temporarily in variables and constants. You learned that variables have a size determined by their type and that the operator `sizeof` can be used to determine the size of one. You got to know of different types of variables such as `bool`, `int`, and so on and that they are to be used to contain different types of data. The right choice of a variable type is important in effective programming, and the choice of a variable that's too small for the purpose can result in a wrapping error or an overflow situation. You learned about the keyword `auto`, where you let the compiler decide the data-type for you on the basis of the initialization value of the variable.

You also learned about the different types of constants and usage of the most important ones among them using the keywords `const`, `constexpr`, and `enum`.

Q&A

Q Why define constants at all if you can use regular variables instead of them?

A Constants, especially those declared using the keyword `const`, are your way of telling the compiler that the value of a particular variable be fixed and not allowed to change. Consequently, the compiler always ensures that the constant variable is never assigned another value, not even if another programmer was to take up your work and inadvertently try to overwrite the value. So, declaring constants where

you know the value of a variable should not change is a good programming practice and increases the quality of your application.

Q Why should I initialize the value of a variable?

A If you don't initialize, you don't know what the variable contains for a starting value. The starting value is just the contents of the location in the memory that are reserved for the variable. Initialization such as that seen here:

```
int myFavoriteNumber = 0;
```

writes the initial value of your choosing, in this case 0, to the memory location reserved for the variable `myFavoriteNumber` as soon as it is created. There are situations where you do conditional processing depending on the value of a variable (often checked against nonzero). Such logic does not work reliably without initialization because an unassigned or initiated variable contains junk that is often nonzero and random.

Q Why does C++ give me the option of using `short int` and `int` and `long int`? Why not just always use the integer that always allows for the highest number to be stored within?

A C++ is a programming language that is used to program for a variety of applications, many running on devices with little computing capacity or memory resources. The simple old cell phone is one example where processing capacity and available memory are both limited. In this case, the programmer can often save memory or speed or both by choosing the right kind of variable if he doesn't need high values. If you are programming on a regular desktop or a high-end smart-phone, chances are that the performance gained or memory saved in choosing one integer type over another is going to be insignificant and in some cases even absent.

Q Why should I not use global variables frequently? Isn't it true that they're usable throughout my application and I can save some time otherwise lost to passing values around functions?

A Global variables can be read and assigned globally. The latter is the problem as they can be changed globally. Assume you are working on a project with a few other programmers in a team. You have declared your integers and other variables to be global. If any programmer in your team changes the value of your integer inadvertently in his code—which even might be a different .CPP file than the one you are using—the reliability of your code is affected. So, sparing a few seconds or minutes should not be criteria, and you should not use global variables indiscriminately to ensure the stability of your code.

Q C++ is giving me the option of declaring unsigned integers that are supposed to contain only positive integer values and zero. What happens if I decrement a zero value contained in an `unsigned int`?

A You see a wrapping effect. Decrementing an unsigned integer that contains 0 by 1 means that it wraps to the highest value it can hold! Check Table 3.1—you see that an `unsigned short` can contain values from 0 to 65,535. So, declare an `unsigned short` and decrement it to see the unexpected:

```
unsigned short myShortInt = 0; // Initial Value
myShortInt = myShortInt - 1; // Decrement by 1
std::cout << myShortInt << std::endl; // Output: 65535!
```

Note that this is not a problem with the `unsigned short`, rather with your usage of the same. An unsigned integer (or short or long) is not to be used when negative values are within the specifications. If the contents of `myShortInt` are to be used to dynamically allocate those many number of bytes, a little bug that allows a zero value to be decremented would result in 64KB being allocated! Worse, if `myShortInt` were to be used as an index in accessing a location of memory, chances are high that your application would access an external location and would crash!

3

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 E, and be certain that you understand the answers before continuing to the next lesson.

Quiz

1. What is the difference between a signed and an unsigned integer?
2. Why should you not use `#define` to declare a constant?
3. Why would you initialize a variable?
4. Consider the `enum` below. What is the value of `Queen`?

```
enum YourCards {Ace, Jack, Queen, King};
```

5. What is wrong with this variable name?

```
int Integer = 0;
```

Exercises

1. Modify `enum YourCards` in quiz question 4 to demonstrate that the value of `Queen` can be 45.
2. Write a program that demonstrates that the size of an unsigned integer and a normal integer are the same, and that both are smaller in size than a long integer.
3. Write a program to calculate the area and circumference of a circle where the radius is fed by the user.
4. In Exercise 3, if the area and circumference were to be stored in integers, how would the output be any different?
5. **BUGBUSTERS:** What is wrong in the following initialization:

```
auto Integer;
```

Index

Symbols

+ (addition) operator, 88–89, 347–349

**+= (addition assignment)
operator, 442–443**

<> (angle brackets), 19

**= (assignment) operator, 87,
357–360**

\ (backslash), 76–78, 86

& (bitwise AND) operator,
100–102, 624

>> (bitwise right shift) operator,
102-104

~ (bitwise NOT) operator,
100-102

| (bitwise OR) operator,
100–102, 624

^ (bitwise XOR) operator,
100–102, 624

{ } (braces), 48, 87

: (colon), 10, 232

```
// comment syntax, 23, 28
```

`/* */` comment syntax, 23, 28

?: (conditional) operator,
126–127

-- (decrement) operator, 89,
190–193, 338–341

* (dereferencing) operator,
183–185, 344–345, 635,
639

. (dot) operator, 218–219

/ (division) operator, 88–89

... (ellipses), 415

== (equality) operator, 92, 352

>> (extraction) operator, 27, 624, 649, 663–664

> (greater than) operator,
92-94, 354-357

>= (greater than or equal to)
operator, 354–357

!= (inequality) operator,
92, 352

++ (increment) operator, 89, 191–193, 338–341, 26, 624, 649, 662, 624

**< (less than) operator, 92–94,
354–357**

<= (less than or equal to)
operator, 92–94, 354–357

&& (logical AND) operator,
95-100

|| (logical OR) operator,
95–100

! (logical NOT) operator,
95-100

-> (member selection) operator,
219–220, 344–345, 635,
639

% (modulo) operator, 88–89

* (multiplication) operator,
88–89

() (parentheses), 364–365,
398–399

& (referencing) operator,
179–180

>>= (right shift) operator, 624

[] (subscript) operator,
197–198, 360–364, 462,
555

~ (tilde), 234
 ""'s operator, 451–452
 :: (scope resolution) operator, 225
 ; (semicolon), 12, 86, 216
 ' (single quotation mark), 45
 - (subtraction) operator, 88–89, 347–349
 0x prefix, 704
 \0 string-terminating character, 76–78, 82

A

abstract base classes, 318–320, 332
 access-specifier, 274
 adapters, container, 425
 adaptive containers. *See* queues; stacks
 adaptive function objects, 538
 addition assignment (+=) operator, 442–443
 addition operator (+), 88–89, 347–349
 addresses of variables
 determining, 179–180
 storing in pointers, 180–182
 adjacent_find(), 571
 advantages of C++, 6
 aggregate initialization, 263–266
 aggregation, 296
 algorithms. *See also* containers; *specific algorithms (for example, for_each())*
 defined, 570, 618
 do's and don'ts, 599
 explained, 426
 interaction with containers, 427–429
 mutating algorithms, 571–573
 non-mutating algorithms, 570–571
 transformations, 585–588

allocation of memory. *See* dynamic memory allocation
 American Standard Code for Information Interchange (ASCII) codes, 41, 711–715
 AND operator
 bitwise AND (&), 100–102, 624
 logical AND (&&), 95–100
 angle brackets (<>), 19
 app class, 661
 app constant, 661
 append(), 442–443
 Area(), 152–153, 158–159
 arguments
 arrays of values, 165–166
 default values, 157–159
 defined, 19, 154
 multiple parameters, 155–156
 no parameters, 156–157
 passing by reference, 166–168, 208–209
 arithmetic operators, 88–89, 347–349
 array class, 699
 array operator ([]), 197–198
 arrays. *See also* string class
 accessing data in, 67–68
 compared to pointers, 195–198
 defined, 64
 do's and don'ts, 71
 dynamic arrays, 66, 74–76, 469–472
 explained, 65
 modifying data in, 69–71
 multidimensional arrays, 71–73, 145–146
 need for, 64–65
 passing to functions, 165–166
 size of, 82–83
 static arrays, 65–66
 storing data in, 66–67

ASCII (American Standard Code for Information Interchange) codes, 41, 711–715
 assert() macro, 399–400
 assignment operator (=), 87, 357–360. *See also* compound assignment operators
 associative containers, 423–424
 ate constant, 661
 auto keyword, 12–13, 48–50, 171–172, 493
 auto_ptr class, 640–643
 automatic type deduction, 48–50, 171–172, 698–699

B

back(), 611
 backslash (\), 76–78, 86
 bad_alloc class, 679
 bad_cast class, 679
 base class methods
 hiding, 286
 initializing, 279–281
 invoking, 283–286
 overridden methods, 281–284
 basic_string class, 450–451
 begin(), 481–482
 best practices, 693–694
 bidirectional iterators, 426
 binary constant, 661
 binary files, reading/writing, 664–665
 binary functions, 545–550
 binary literals, 51
 binary numeral system, 702–703, 705–706
 binary operators
 arithmetic operators, 347–349
 compound assignment operators, 350–352
 copy assignment, 357–360
 equality/inequality, 352

- explained, 346
 - function `()`, 364–365
 - move assignment, 365–371
 - move constructor, 365–371
 - relational operators, 354–357
 - subscript `[]`, 360–364
 - binary predicates, 363, 547–550**
 - binary_search(), 573, 592–595**
 - bits, 703**
 - bitset class, 622–627, 706**
 - bitwise operators, 100–104, 624**
 - blocks, 87, 117–118**
 - bool type, 40**
 - Boolean variables, declaring, 40**
 - Boost Thread Libraries, 645, 690**
 - braces `{ }`, 48, 87**
 - break statement, 139–140**
 - breakdown method, 705**
 - buckets, 507**
 - bugs, 8**
 - bytes, 703**
- C**
- .c filename extension, 9**
 - c_str(), 440**
 - C++**
 - advantages of, 6
 - evolution of, 7
 - history of, 6
 - new features, 12–13
 - revisions of, 12–13
 - C++17, 695–699**
 - capture lists, 559–560**
 - case, converting, 449–450**
 - case sensitivity, 20, 398**
 - casting**
 - `const_cast` operator, 385–386
 - C-style casts, 379
 - do's and don'ts, 388
 - `dynamic_cast` operator, 381–384
 - explained, 377
 - need for, 378
 - problems with, 386–387
 - `reinterpret_cast` operator, 384–385
 - `static_cast` operator, 380–381
 - upcasting, 380
 - catch keyword, 673–675**
 - catching exceptions**
 - all exceptions, 673–674
 - catch keyword, 673–675
 - example, 677–679
 - exceptions of type, 674–675
 - failed memory allocation, 202–204
 - try keyword, 673
 - cbegin(), 442**
 - cerr class, 651**
 - char buffer, writing to, 657–658**
 - char type, 41**
 - character variables, declaring, 41**
 - chunking separators, 45**
 - cin class, 26–28, 651, 656–660**
 - Circumference(), 152–153**
 - classes. *See also* constructors; destructors; inheritance; individual classes (for example, *fstream* class)**
 - accessing members of, 218–220
 - aggregate initialization, 263–266
 - aggregation, 296
 - compared to structs, 257–258
 - composition, 296
 - `constexpr` keyword, 266–267
 - declaring, 216–217
 - deep copying, 240–244
 - explained, 216
 - friend classes, 258–260
 - instantiating, 217–218
 - instantiation on stack, prohibiting, 249–251
 - naming conventions, 219
 - non-copyable objects, ensuring, 246
 - private keyword, 220–224
 - public keyword, 220–222
 - shallow copying, 237–240
 - singleton classes, 247–249
 - `sizeof()` on, 255–257
 - subclasses, 275
 - super classes, 275
 - unions, 260–266
 - clear(), 472, 483**
 - Clone(), 638**
 - CodeGuru, 699**
 - CodeProject, 699**
 - collections, inserting elements into, 597–599**
 - collisions, 529**
 - colon `:`, 10**
 - comments, 18, 23, 28**
 - compilation, 8. *See also* preprocessor directives**
 - compilers, 10–12, 13, 709–710
 - compile-time checks, 417–418
 - compile-time errors, 14
 - conditional code compilation, 697–698
 - example, 10–12
 - operator precedence, 709–710
 - compiled languages, 13**
 - complexity, 424**
 - composition, 296**
 - compound assignment operators, 104–106, 350–352**
 - compound statements, 87, 117–118**
 - concatenation, 79–81, 442–443**
 - conditional code compilation, 697–698**

conditional operator (?:),
126–127

conditional programming

conditional operator (?:),
126–127

if statement, 114–122

loops. *See* loops

switch-case statement,
122–125

const keyword, 52–53,
193–194, 208, 363

const_cast operator, 385–386

in constant, 661

constant complexity, 424

constant expressions, 53–55

constants. *See also* variables

constant expressions, 53–55

declared constants, 52–53

defined, 50–51

defining, 57, 59–60, 392–394

enumerators, 55–57

literal constants, 51–52,
371–372

naming conventions, 58–59

constexpr keyword, 53–55,
233, 266–267

construction, order of, 288

constructors

automatic type deduction,
698–699

classes without default
constructors, 228–230

converting constructors,
251–253

copy constructors, 233–244

declaring, 224–225

default constructors, 228–230

default values, 230–231

initialization lists, 231–233

move constructors, 244–245

order of construction, 288

overloading, 227–228

uses for, 246–253

virtual copy constructors,
328–331

when to use, 225–226

containers

associative containers,
423–424

choosing, 429–431

container adapters, 425

defined, 422

elements

copy and remove
operations, 588–590

counting, 576–577

finding, 573–575

initializing, 580–583

inserting into collections,
597–599

partitioning, 595–597

processing, 583–585

replacing, 590–592

searching for, 577–579

sorting, 592–595

interaction with algorithms,
427–429

queues, 604–618

sequential containers,
422–423

stacks, 604–608

continue statement, 139–140

conventional pointers, 634

conversion operators, 341–343

converting constructors,
251–253

copy(), 572, 588–590

copy assignment operator,
357–360

copy constructors, 233–244

copy elision, 696

Copy on Write (COW), 639

copy_backward(), 572,
588–590, 600

copy_if(), 588–590

count(), 500, 570, 576–577,
625

count_if(), 570, 576–577

cout class, 651, 653–656, 706

cout statement, 20

COW (Copy on Write), 639

.cpp filename extension, 8

cppreference.com, 699

CPUs, multicore, 688–689

custom sort predicates,
525–528

CustomException class,
680–682

D

dangling pointers, 200–201

deadlock, 692

debugging, 8

decimal numeral system, 702,
705–706,

decrement operator (--), 89,
191–193, 338–341

deepcopy_smart_ptr class,
637–639

deep-copy-based smart
pointers, 637–639

default constructors, 228–230

default template parameters,
407–408

default values, function
parameters with, 157–159

#define directive, 57

defining constants with,
392–394

writing macro functions with,
396–398

definitions (function), 154

delete operator, 187–190

DemoConsoleOutput(), 24–26

deque class, 422, 469–472,

dereference operator (*),
344–345

dereferencing operator (*),
183–185, 635, 639

derivation, 272–276

DerivedFunction(), 380

destruction, order of, 288–290

destructive copy smart
pointers, 640–643

destructors, 635

- declaring, 233–234
- order of destruction, 288–290
- private destructors, 249–251
- virtual destructors, 310–314
- when to use, 234–237

directives. See preprocessor directives**display number formats, changing, 653–655****DisplayArray(), 165–166****DisplayComparison(), 405****DisplayContents(), 500****DisplayElementKeepCount(), 541–543****DisplayTupleInfo(), 416–417****DisplayVector(), 467****division operator (/), 88–89****do...while loop, 132–133****documentation, 699****dot operator (.), 218–219****double type, 45****dynamic arrays. See also vector class**

- declaring, 74–76

- defined, 66

- deque class, 469–472

dynamic memory allocation

- delete operator, 187–190

- explained, 187

- failed memory allocation, 202–204

- new operator, 187–190

dynamic_cast operator, 381–384**E****Eclipse IDE, 8****ellipses (...), 415****empty(), 472, 611, 615****empty return values, 156–157****end(), 481–482****#endif directive, 395–396****endl manipulator, 652****ends manipulator, 652****enumerations, 55–57****enumerators, 55–57****equal(), 571****equality operator (==), 92, 352****erase(), 445–447, 482–483, 502–506, 522–524****errors. See also exceptions**

- compiler errors, 12

- compile-time errors, 14

- debugging, 8

- fence-post errors, 70

- narrowing conversion errors, 48

- overflow errors, 43–44

- runtime errors, 14

evolution of C++7**exception class, 679–680****exceptions. See also errors**

- catching, 673–679

- causes of, 672

- custom classes, 680–682

- defined, 671

- do's and don'ts, 682

- std::exception class, 679–680

- throwing, 676–679, 683

exclusive OR operator, 95–96**executables, generating, 7–8****explicit keyword, 252–253****explicit type declaration, 493****expressions. See also lambda expressions**

- constant, 53–55

- validating, 399–400

extraction operator (>>), 27, 624, 649, 663–664**F****failed memory allocation, 202–204****false value, 94****fence-post errors, 70****Fibonacci number calculation, 145–146****Fibonacci numbers, calculating, 159–161****field width, setting, 655–656****filename extensions, 8, 9****fill(), 571, 580****fill_n(), 571, 580****final specifier, 300, 327, 708****find(), 426, 444–445, 500–502, 519–522, 571, 573–575****find_end(), 571****find_first_of(), 571****find_if(), 426, 558, 571, 573–575****fixed manipulator, 652****fixed-width integer types, 47****flip(), 625, 628–629****float type, 45****floating-point variables, 45****flow control. See program flow, controlling****for_each(), 554, 555–556, 572, 583–585****forward iterators, 425–426****forward slash (/), 88–89****forward_list class, 422, 490–492****friend classes, 258–260****front(), 611****fstream class, 651**

- opening/closing files, 660–661

- reading binary files, 664–665

- reading text files, 663–664

- writing to binary files, 664–665

- writing to text files, 662–663

FuncDisplayElement(), 538–541**function objects**

- binary functions, 545–550, 562–563

- defined, 528

explained, 538

unary functions, 538–545,
555–557

function operator, 364–365

function prototypes, 153–154

<functional>615

**functions. *See also* arguments;
function objects; macros;
virtual functions; *specific
functions (for example,
count())***

automatic return type
deduction, 171–172

definitions, 154

explained, 23–26

inline functions, 169–171

microprocessor-level
implementation of, 168–169
multiple return statements,
161

need for, 152–153

overloading, 163–164

overridden functions,
preventing, 327

passing pointers to, 194–195
pointers, storing addresses in,
180–182

prototypes, 153–154

template functions, 403–405

functors. *See* function objects

G

g++ compiler, 8, 10–11

generate(), 571, 581–583

generate_n(), 581–583

get(), 668

GetFibNumber(), 159–161

GetInstance(), 249

getline(), 28, 659–660, 668

GetMax(), 403–405

GetPi(), 53–54, 169–171

GHz (gigahertz), 688

gigabytes, 704

global variables, 37–38, 60

goto loop, 128–130

**greater than (>) operator,
92–94, 354–357**

**greater than or equal to (>=)
operator, 354–357**

**grouped if...else constructs,
121–122**

H

**handling exceptions. *See*
exceptions**

hash sets, 507–509

hash tables, 528–533

HashFunction(), 529

**header files, inclusion guards
in, 419**

Hello World program, 9–11, 18

hex manipulator, 652

**hexadecimal numbering
system, 178, 704–705,
706–706**

history of C++6

HoldsPair template, 408–409

Hungarian notation, 38–39

Hz (hertz), 688

I

**IDEs (Integrated Development
Environments), 8–9, 38–39**

**if statement, 114–122,
695–696**

#ifndef directive, 395–396

ifstream class, 651

implicit conversions, 343

#include directive, 18–19, 28

**inclusion guards, 395–396,
419**

**increment operator (++), 89,
191–193, 338–341**

**inequality (!=) operator, 92,
352**

infinite loops, 141–143

inheritance

avoiding with final, 300

base class methods, 279–286

do's and don'ts, 301

multiple inheritance, 297–300

order of construction, 288

order of destruction, 288–290

private inheritance, 291

protected, 276–279

protected inheritance,
293–296

public, 273–274

slicing, 297

initialization lists, 231–233

initializer statement, 695–696

inline functions, 169–171

inline keyword, 169–171

input iterators, 425

**input/output, 26–28. *See also*
streams**

**insert(), 459–462, 479–482,
499, 608**

**insertion operator (<<), 26,
624, 649, 662**

int keyword, 19–20

int type, 42

int8_t type, 47

int16_t type, 47

int32_t type, 47

int64_t type, 47

integers, 60

fixed-width, 47

signed, 41–42

unsigned, 61

**Integrated Development
Environments (IDEs), 8–9,
38–39**

**International Organization for
Standardization (ISO), 6, 11**

interpreted languages, 13

**intrusive reference counting,
639**

invalid memory locations, 199

<iomanip> manipulators, 652

iostream file, 18

ISO (International Organization for Standardization), 6, 11

iterative statements. *See* loops

iterators, 425–426, 429, 534–535, 630

J-K-L

keywords, list of, 58–59, 707–708. *See also* individual keywords (for example, *auto*)

kilobytes, 704–704

lambda expressions

for binary functions, 562–563

for binary predicates, 564–566

in C++17, 698

do's and don'ts, 566

explained, 12–13, 172–174, 554–555

local variables in, 567

state maintenance, 559–560, 567

syntax, 560–562

for unary functions, 555–557

for unary predicates, 557–558

leaks, memory, 188, 198–204

left shift operator (<<), 102–104

length of strings, determining, 79–81

less, 522–524

less than (<) operator, 92–94, 354–357

less than or equal to (<=) operator, 92–94, 354–357

lexicographical_compare(), 571

line breaks, 12

linear complexity, 424

lists, 422

capture lists, 559–560

do's and don'ts, 492

erasing elements from, 482–483

initialization, 48, 459

initialization lists, 231–233

inserting elements into, 478–482

instantiating, 476–477

removing elements from, 487–490

reversing elements in, 484–485

sorting elements in, 485–486

literal constants, 51–52, 371–372

load_factor(), 509, 530

local variables in lambda expressions, 567

logarithmic complexity, 424

logical AND (&&) operator, 95–100

logical NOT (!) operator, 95–100

logical OR (||) operator, 95–100

logical XOR operator, 95–96

long long type, 42

long type, 42

for loop, 133–139

loops

access arrays with, 71

do...while, 132–133

exiting, 139–140

for, 133–139

goto, 128–130

infinite, 141–143

nested, 143–148

resuming execution of, 139–140

while, 130–132

lower_bound(), 573, 597–599

lowercase, converting strings to, 449–450

l-values, 87–88

M

macros

advantages/disadvantages, 400–401

assert(), 399–400

case sensitivity, 398

compared to templates, 419

do's and don'ts, 401

inclusion guards in, 395–396

parentheses in, 398–399

protecting against multiple inclusion with, 395–396

tuples, 415–417

writing, 396–398

main(), 19–20

manipulators, stream, 651–652

map class, 423

custom sort predicates, 525–528

erasing elements from, 522–524

explained, 514

finding elements in, 519–521

inserting elements into, 517–519

instantiating, 515–516

max_bucket_count(), 509

max_load_factor(), 509

megabytes, 704

megahertz (Mhz), 688

member selection operator (->), 344–345, 635, 639

memory. *See also* pointers

accessing with variables. *See* variables

dynamic memory allocation, 187–190, 202–204

l-values, 87–88

memory leaks, 188, 198–204

RAM (Random Access Memory), 32

raw pointers, 634

r-values, 87–88

- stack, 169
- variable addresses, determining, 179–180
- memory leaks, 188, 198–204**
- methods. See functions**
- Mhz (megahertz), 688**
- Microsoft Visual Studio Express, 8**
- MIN macro, 397, 400–401**
- mismatch(), 571**
- modulo operator (%), 88–89**
- move assignment operator, 365–371**
- move constructor operator, 365–371**
- move constructors, 244–245**
- multicore processors, 688–689**
- multidimensional arrays, 71–74, 145–146**
- multimap class, 423**
 - custom sort predicates, 525–528
 - erasing elements from, 522–524
 - finding elements in, 522
 - inserting elements into, 517–519
 - instantiating, 515–516
- multiple inclusion, protecting against, 395–396**
- multiple inheritance, 297–300**
- multiplication operator (*), 88–89**
- MultiplyNumbers(), 35–36**
- multiset class, 423, 496–510**
 - do's and don'ts, 510
 - erasing elements in, 502–506
 - explained, 496
 - finding elements in, 500–502
 - inserting elements into, 499–500
 - instantiating, 497–498
 - pros and cons of, 507
- multithreading, 690–693**
- mutating algorithms, 571–573**
- mutex class, 692**

N

- [N] operator, 624**
- namespaces, 21–22**
- naming conventions**
 - classes, 219
 - constants, 58–59
 - variables, 34, 38–39, 58–59
- narrowing conversion errors, 48**
- nested if statements, 118–122**
- nested loops, 143–148**
- .NET, 6**
- new operator, 187–190, 202–204**
- new(nothrow) operator, 204, 210**
- non-copyable objects, ensuring, 246**
- non-mutating algorithms, 570–571**
- NOT operator**
 - bitwise NOT (~), 100–102
 - logical NOT (!), 95–100
- null terminator, 76–78, 82**
- numeric codes (ASCII), 41, 711–715**

O

- .o filename extension, 8**
- .obj filename extension, 8**
- objects, creating, 217–218**
- oct manipulator, 652**
- octal numeral system, 705**
- OFF state, 702**
- ofstream class, 651**
- online communities, 699**
- online documentation, 699**
- open(), 660, 662–664**
- operator keyword, 336**
- operators. See also binary operators; casting; overloaded operators; unary operators**

- declaring, 336
- do's and don'ts, 374
- explained, 336–337
- precedence of, 108–109, 709–710
- prefix versus postfix, 90–91
- order of construction, 288**
- order of destruction, 288–290**
- OR operator**
 - bitwise OR (|), 100–102, 624
 - logical OR (||), 95–100
- out constant, 661**
- output, 26–28. See also streams**
- output iterators, 425**
- overflow errors, 43–44**
- overloaded constructors, 227–228**
- overloaded functions, 163–164**
- overloaded operators, 635**
 - copy assignment, 357–360
 - equality/inequality, 352
 - operators that cannot be overloaded, 373
 - relational operators, 354–357
- overridden functions, 281–284, 327**
- override specifier, 326–327, 708**

P

- partial_sort(), 572**
- partial_sort_copy(), 573**
- partition(), 573, 595–597**
- POD (Plain Old Data), 45, 656–657**
- pointer operator (->), 219–220, 344–345**
- pointers. See also smart pointers**
 - accessing data in, 183–185
 - compared to arrays, 195–198
 - const keyword, 193–194

- declaring, 178–179, 180
- decrement operator (--), 191–193
- defined, 178
- do's and don'ts, 205
- dynamic memory allocation, 187–190
- increment operator (++), 191–193
- passing to functions, 194–195
- raw pointers, 634
- sizeof(), 185–186
- this, 254
- polymorphism. *See also* virtual functions**
 - abstract base classes, 318–320, 332
 - defined, 306
 - do's and don'ts, 331
 - need for, 306–308
- pop(), 607–608, 611–613, 616–618**
- pop operations, 169**
- pop_back(), 465–466, 470–472**
- pop_front(), 470–472**
- postfix operators, 90–91**
- precedence of operators, 108–109**
- predicates, 363**
 - binary predicates, 547–550
 - custom sort predicates, 522–524
 - unary predicates, 543–545, 557–558
- prefix, 21–22**
- prefix operators, 90–91**
- preprocessor, 392**
- preprocessor directives**
 - #define, 57, 392–398
 - defined, 18
 - #endif, 395–396
 - explained, 392
 - #ifndef, 395–396
 - #include, 18–19
- printable characters (ASCII), 41, 711–715**
- priority_queue class, 425, 613–618**
- private destructors, 249–251**
- private inheritance, 291**
- private keyword, 220–224, 291**
- processors, multicore, 688–689**
- program flow, controlling. *See also* loops**
 - conditional operator (?:), 126–127
 - if statement, 114–122
 - switch-case statement, 122–125
- programs**
 - building, 10–11
 - compiling, 8, 10–12
 - executables, 7–8
 - executing, 10–11
 - structure of, 18–28
- protected inheritance, 276–279, 293–296**
- protected keyword, 276–279, 293**
- prototypes (function), 153–154**
- public inheritance, 273–274**
- public keyword, 220–222, 273–274, 527**
- pure virtual functions, 318–320**
- push(), 607–608, 611–613, 616–618**
- push operations, 169**
- push_back(), 76, 458, 470–472, 478–479, 608, 629**
- push_front(), 470–472, 478–479**
- Q-R**
- queues**
 - explained, 425, 604–605
 - operators, 628–629
 - priority_queue class, 613–618
 - queue class, 425, 609–613
- race conditions, 692**
- RAM (Random Access Memory), 32**
- random access iterators, 426**
- range-based for loops, 137–139**
- raw pointers, 634**
- read(), 664–665**
- recursion, 159–161**
- references**
 - const keyword, 208
 - explained, 205–206
 - intrusive reference counting, 639
 - need for, 206–208
 - passing by reference, 166–168, 208–209
 - reference-counted smart pointers, 639–640
 - reference-linked smart pointers, 640
- referencing operator (&), 179–180**
- refinements, 426**
- reinterpret_cast operator, 384–385**
- relational operators, 92–94, 354–357**
- remove(), 572, 590**
- remove_copy(), 572**
- remove_if(), 426, 550, 572, 590**
- replace(), 572, 590–592**
- replace_if(), 572, 590–592**
- reserve(), 473**
- reserved words. *See* keywords, list of**
- reset(), 625**
- resetioflags(), 653–654**
- resetiosflag manipulator, 652**
- resuming loop execution, 139–140**
- return statements, multiple, 161**
- return values, 20**

reverse(), 448–449, 484–485
reverse algorithm, 426
right shift operator (>>), 102–104
RTTI (runtime type identification), 318, 382
runtime errors, 14
r-values, 87–88

S

safety (type), templates and, 405
scientific manipulator, 652
scope of variables, 35–36
scope resolution (::) operator, 225
search(), 570, 577–579
search_n(), 570, 577–579
semaphores, 692
separators, chunking, 45
sequential containers, 422–423
set(), 625
set class
 do's and don'ts, 510
 erasing elements in, 502–506
 explained, 423, 496–510
 finding elements in, 500–502
 inserting elements into, 499–500
 instantiating, 497–498
 pros and cons of, 507
setbase manipulator, 652
setfill(), 652, 655–656
setiosflags(), 652, 653–654
setprecision(), 652
setw(), 652, 655–656
shallow copying, 237–240
shared_ptr class, 645
shift operators, 102–104
short type, 42
signed integers, 41–42

single quotation mark ('), 45
singleton classes, 247–249
sizeof(), 611, 615, 625
sizeof(), 46–47, 106–107, 185–186, 255–257
slicing, 297, 637–638
smart pointers

 advantages of, 634–635
 auto_ptr, 640–643
 COW (Copy on Write), 639
 deep copy, 637–639
 defined, 344, 634
 destructive copy, 640–643
 implementing, 635–636
 libraries, 645
 reference-counted, 639–640
 reference-linked, 640
 shared_ptr, 645
 slicing issues, 637–638
 unique_ptr, 643–645
 weak_ptr, 645

sort(), 485–486, 547–550, 564–566, 572, 592–595, 600

specialization (templates), 410–411

SQUARE macro, 396–397

stable_partition(), 573, 595–597

stable_sort(), 572, 595

StackOverflow, 699

stacks, 169, 604

 inserting and removing elements in, 607–608
 instantiating, 605–606
 member functions, 606–607
 prohibiting instantiation on, 249–251
 stack class, 425, 605–608

ON state, 702

statements. See also loops

 blocks, 87, 117–118
 break, 139–140
 continue, 139–140

 cout, 20
 explained, 86
 if, 114–122
 initializer, 695–696
 return, 161
 switch, 695–696
 switch-case, 122–125

static arrays, 65–66

static keyword, 247–248

static members, 412–413

static_assert, 417–418

static_cast operator, 380–381

std namespace, 21–22

stray pointers, 200–201

strcat(), 79

strcpy(), 79

streams

 cin class, 656–660
 classes, 651
 cout class, 653–656
 explained, 649–650
 extraction operator (>>), 649
 fstream class, 660–665
 insertion operator (<<), 649
 manipulators, 651–652
 stringstream class, 665–668
string class, 79–81, 432
 “”s operator in, 451–452
 accessing, 440–442
 concatenating, 442–443
 converting case of, 449–450
 finding characters in, 444–445
 instantiating, 437–440
 need for, 436–437
 reversing contents of, 448–449
 template-based implementation, 450–451
 truncating, 445–447
 writing to, 658–660

string literals, 20

string_view class, 452, 696

stringstream class, 651, 665–668
 strlen(), 79
 Stroustrup, Bjarne, 6
 structs, 257–258
 subclasses, 275
 subscript ([]) operator, 360–364
 subscript operator ([]), 462
 substrings, finding, 444–445
 subtraction operator (-), 88–89, 347–349
 subtype polymorphism. *See* polymorphism
 super classes, 275
 SurfaceArea(), 155–156
 switch statement, 695–696
 switch-case statement, 122–125
 synchronization, thread, 691, 692

T

tables

hash tables, 528–533
 virtual function table, 314–318

template keyword, 397

template<auto>699

templates

compared to macros, 419
 compile-time checks, 417–418
 declaring, 402–408
 do's and don'ts, 418
 explained, 402
 HoldsPair example, 408–409
 instantiating, 410–411
 programming applications of, 418
 specialization, 410–411
 static members, 412–413
 template classes, 406–407
 variable templates, 413–417

terabytes, 704

text files

closing, 660–661
 opening, 660–661
 reading, 663–664
 writing to, 662–663

this pointer, 254

threads, 689–693

throw keyword, 676–677

throwing exceptions, 676–679, 683

tilde (~), 100–102

top(), 615

transform(), 426, 449–450, 545–547, 562–563, 572, 585–588

transformations, 585–588

true value, 94

trunc constant, 661

try keyword, 673

tuple class, 415–417

TwicePi(), 53–54

typedef keyword, 50

types

automatic type deduction, 48–50, 698–699
 bool, 40
 casting, 377–388
 char, 41
 conversion, 251–253
 double, 45
 exceptions of type, handling, 674–675
 explicit type declaration, 493
 fixed-width integer types, 47
 float, 45
 int, 42
 iterator type definitions, 429
 long, 42
 long long, 42
 overflow errors, 43–44
 RTTI (runtime type identification), 382
 short, 42

table of, 39–40

type safety, templates and, 405

type substitution, 50

unsigned int, 42–43

unsigned long, 42–43

unsigned long long, 42–43

unsigned short, 42–43

U

uint8_t type, 47

uint16_t type, 47

uint32_t type, 47

uint64_t type, 47

unary functions, 538–545

unary operators

conversion operators, 341–343
 dereference, 344–345
 explained, 337–338
 increment/decrement, 89, 338–341
 logical operators, 95–100
 member selection, 344–345

unary predicates, 363, 543–545, 557–558

unions

aggregate initialization, 263–266
 declaring, 260–263
 typesafe alternative to, 697
 when to use, 261–263

unique(), 572

unique_copy(), 572

unique_ptr class, 643–645

unordered_map class, 423, 528–533

unordered_multimap class, 423, 528–533

unordered_multiset class, 423, 507–509

unordered_set class, 423, 507–509

unsigned int type, 42–43
unsigned integers, 42–43, 61
unsigned long long type, 42–43
unsigned long type, 42–43
unsigned short type, 42–43
upcasting, 380
upper_bound(), 573, 597–599
uppercase, converting strings to, 449–450
User Interface Threads, 690
user-defined literals, 52, 371–372
using keyword, 21–22

V

values

default values, 157–159
 passing to functions
 arrays of values, 165–166
 by reference, 166–168

variable templates, 413–417

variables. *See also* constants; types

addresses
 determining, 179–180
 storing in pointers, 180–182
 declaring
 automatic type inference, 48–50
 Boolean variables, 40
 character variables, 41
 floating-point variables, 45

multiple variables, 34
 overflow errors, 43–44
 signed integers, 42
 single variables, 32–34
 unsigned integers, 42–43
 global variables, 37–38, 60
 initializing, 60
 integers, 60
 list initialization, 48
 local variables, 567
 naming conventions, 38–39, 58–59
 narrowing conversion errors, 48
 scope, 35–36
 size of, 46–47

variadic templates, 413–417

variant class, 263, 697

vector class, 74–76, 422

accessing elements in, 462–465
 characteristics of, 455–456
 do's and don'ts, 472
 initialization, 459
 inserting elements into, 458–462
 instantiating, 456–457
 removing elements from, 465–466
 size and capacity of, 467–469

vector<bool>

accessing with iterators, 630
 explained, 627
 functions, 628–629

instantiating, 627–628
 specifying number of elements in, 630

virtual copy constructors, 328–331

virtual destructors, 310–314

virtual function table, 314–318

virtual functions

abstract base classes, 318–320
 declaring, 308–310
 do's and don'ts, 331
 final specifier, 327
 inheritance, 321–325
 override specifier, 326–327
 pure virtual functions, 318–320
 virtual function table, 314–318

virtual inheritance, 321–325

virtual keyword, 318, 324–325, 332

W-X-Y-Z

warning messages, 13

weak_ptr class, 645

while loop, 130–132

wild pointers, 200–201

Worker Threads, 690

write(), 664–665

wstring class, 432, 450–451

XOR operator

bitwise XOR, 100–102, 624
 logical XOR, 95–96