

Rogers Cadenhead  
Jesse Liberty

**SIXTH  
EDITION**

Updated  
for C++14

Sams **Teach Yourself**

**C++**

in **24**  
**Hours**

**SAMS**

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



# Contents at a Glance

## Part I: Beginning C++

1	Writing Your First Program	5
2	Organizing the Parts of a Program	13
3	Creating Variables and Constants	27
4	Using Expressions, Statements, and Operators	43
5	Calling Functions	61
6	Controlling the Flow of a Program	79
7	Storing Information in Arrays and Strings	95

## Part II: Classes

8	Creating Basic Classes	109
9	Moving into Advanced Classes	123

## Part III: Memory Management

10	Creating Pointers	135
11	Developing Advanced Pointers	155
12	Creating References	167
13	Developing Advanced References and Pointers	183

## Part IV: Advanced C++

14	Calling Advanced Functions	199
15	Using Operator Overloading	213

## Part V: Inheritance and Polymorphism

16	Extending Classes with Inheritance	231
17	Using Polymorphism and Derived Classes	251
18	Making Use of Advanced Polymorphism	267

## Part VI: Special Topics

19	Storing Information in Linked Lists	287
20	Using Special Classes, Functions, and Pointers	301
21	Using New Features of C++14	331
22	Employing Object-Oriented Analysis and Design	343
23	Creating Templates	373
24	Dealing with Exceptions and Error Handling	391

## Part VII: Appendixes

A	Binary and Hexadecimal	411
B	Glossary	419
C	This Book's Website	427
D	Using the MinGW C++ Compiler on Windows	429

Rogers Cadenhead  
Jesse Liberty

Sams **Teach Yourself**

**C++**

in **24**  
**Hours**

SIXTH EDITION

**SAMS**

800 East 96th Street, Indianapolis, Indiana, 46240 USA

## **Sams Teach Yourself C++ in 24 Hours**

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/](http://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: 978-0-672-33746-8

ISBN-10: 0-672-33746-0

Library of Congress Control Number: 2016945006

First Printing August 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 authors 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 or programs accompanying it.

### **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](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact  
[governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the U.S., please contact  
[intlcs@pearson.com](mailto:intlcs@pearson.com).

### **Editor**

Mark Taber

### **Project Editor**

Lori Lyons

### **Project Manager**

Prashanthi Nadipalli

### **Copy Editor**

Christopher Morris

### **Technical Editor**

Jon Upchurch

# Contents at a Glance

Introduction .....	1
<b>Part I: Beginning C++</b>	
<b>HOUR 1</b> Writing Your First Program .....	5
<b>2</b> Organizing the Parts of a Program .....	13
<b>3</b> Creating Variables and Constants .....	27
<b>4</b> Using Expressions, Statements, and Operators .....	43
<b>5</b> Calling Functions .....	61
<b>6</b> Controlling the Flow of a Program .....	79
<b>7</b> Storing Information in Arrays and Strings .....	95
<b>Part II: Classes</b>	
<b>HOUR 8</b> Creating Basic Classes .....	109
<b>9</b> Moving into Advanced Classes .....	123
<b>Part III: Memory Management</b>	
<b>HOUR 10</b> Creating Pointers .....	135
<b>11</b> Developing Advanced Pointers .....	155
<b>12</b> Creating References .....	167
<b>13</b> Developing Advanced References and Pointers .....	183
<b>Part IV: Advanced C++</b>	
<b>HOUR 14</b> Calling Advanced Functions .....	199
<b>15</b> Using Operator Overloading .....	213
<b>Part V: Inheritance and Polymorphism</b>	
<b>HOUR 16</b> Extending Classes with Inheritance .....	231
<b>17</b> Using Polymorphism and Derived Classes .....	251
<b>18</b> Making Use of Advanced Polymorphism .....	267

**Part VI: Special Topics**

<b>HOUR 19</b>	Storing Information in Linked Lists .....	287
<b>20</b>	Using Special Classes, Functions, and Pointers .....	301
<b>21</b>	Using New Features of C++14 .....	331
<b>22</b>	Employing Object-Oriented Analysis and Design .....	343
<b>23</b>	Creating Templates .....	373
<b>24</b>	Dealing with Exceptions and Error Handling .....	391

**Part VII: Appendixes**

<b>A</b>	Binary and Hexadecimal .....	411
<b>B</b>	Glossary .....	419
<b>C</b>	This Book's Website .....	427
<b>D</b>	Using the MinGW C++ Compiler on Windows .....	429
	Index .....	439

# Table of Contents

Introduction	1
<b>Part I: Beginning C++</b>	
<b>HOUR 1: Writing Your First Program</b>	<b>5</b>
Using C++	5
Compiling and Linking the Source Code	6
Creating Your First Program	7
<b>HOUR 2: Organizing the Parts of a Program</b>	<b>13</b>
Reasons to Use C++	13
The Parts of a Program	17
Comments	19
Functions	20
<b>HOUR 3: Creating Variables and Constants</b>	<b>27</b>
What Is a Variable?	27
Defining a Variable	32
Assigning Values to Variables	33
Using Type Definitions	34
Constants	36
Auto-Typed Variables	38
<b>HOUR 4: Using Expressions, Statements, and Operators</b>	<b>43</b>
Statements	43
Expressions	44
Operators	45
<b>If-Else</b> Conditional Statements	52
Logical Operators	56
Tricky Expression Values	58

<b>HOURL 5: Calling Functions</b>	<b>61</b>
What Is a Function?	61
Declaring and Defining Functions	61
Using Variables with Functions	64
Function Parameters	67
Returning Values from Functions	68
Default Function Parameters	70
Overloading Functions	72
Auto-Typed Return Values	73
<b>HOURL 6: Controlling the Flow of a Program</b>	<b>79</b>
Looping	79
while Loops	79
do-while Loops	83
for Loops	84
switch Statements	89
<b>HOURL 7: Storing Information in Arrays and Strings</b>	<b>95</b>
What Is an Array?	95
Writing Past the End of Arrays	97
Initializing Arrays	98
Multidimensional Arrays	99
Character Arrays	102
Copying Strings	104
Reading Arrays with Foreach Loops	105
<b>Part II: Classes</b>	
<b>HOURL 8: Creating Basic Classes</b>	<b>109</b>
What Is a Type?	109
Creating New Types	110
Classes and Members	110
Accessing Class Members	112
Private Versus Public Access	112
Implementing Member Functions	114
Creating and Deleting Objects	116



<b>HOURL 9: Moving into Advanced Classes</b>	<b>123</b>
<b>const</b> Member Functions .....	123
Interface Versus Implementation .....	124
Organizing Class Declarations and Function Definitions .....	124
Inline Implementation .....	124
Classes with Other Classes as Member Data .....	127
<b>Part III: Memory Management</b>	
<b>HOURL 10: Creating Pointers</b>	<b>135</b>
Understanding Pointers and Their Usage .....	135
The Stack and the Heap .....	145
Null Pointer Constant .....	150
<b>HOURL 11: Developing Advanced Pointers</b>	<b>155</b>
Creating Objects on the Heap .....	155
Deleting Objects .....	155
Accessing Data Members Using Pointers .....	157
Member Data on the Heap .....	158
The <b>this</b> Pointer .....	160
Stray or Dangling Pointers .....	161
<b>const</b> Pointers .....	162
<b>const</b> Pointers and <b>const</b> Member Functions .....	163
<b>HOURL 12: Creating References</b>	<b>167</b>
What is a Reference? .....	167
Creating a Reference .....	167
Using the Address of Operator on References .....	169
What Can Be Referenced? .....	171
Null Pointers and Null References .....	172
Passing Function Arguments by Reference .....	172
Understanding Function Headers and Prototypes .....	177
Returning Multiple Values .....	177
<b>HOURL 13: Developing Advanced References and Pointers</b>	<b>183</b>
Passing by Reference for Efficiency .....	183
Passing a <b>const</b> Pointer .....	186

References as an Alternative to Pointers .....	189
When to Use References and When to Use Pointers .....	191
References to Objects Not in Scope .....	191
Returning a Reference to an Object on the Heap .....	192
Pointer, Pointer, Who Has the Pointer? .....	195

## **Part IV: Advanced C++**

<b>HOUR 14: Calling Advanced Functions</b> .....	<b>199</b>
Overloaded Member Functions .....	199
Using Default Values .....	201
Initializing Objects .....	203
The Copy Constructor .....	204
Compile-Time Constant Expressions .....	208
<b>HOUR 15: Using Operator Overloading</b> .....	<b>213</b>
Operator Overloading .....	213
Conversion Operators .....	224

## **Part V: Inheritance and Polymorphism**

<b>HOUR 16: Extending Classes with Inheritance</b> .....	<b>231</b>
What Is Inheritance? .....	231
Private Versus Protected .....	234
Constructors and Destructors .....	236
Passing Arguments to Base Constructors .....	239
Overriding Functions .....	244
<b>HOUR 17: Using Polymorphism and Derived Classes</b> .....	<b>251</b>
Polymorphism Implemented with Virtual Member Functions .....	251
How Virtual Member Functions Work .....	256
<b>HOUR 18: Making Use of Advanced Polymorphism</b> .....	<b>267</b>
Problems with Single Inheritance .....	267
Abstract Data Types .....	271

**Part VI: Special Topics**

<b>HOUR 19: Storing Information in Linked Lists</b>	<b>287</b>
Linked Lists and Other Structures .....	287
Linked List Case Study .....	289
Linked Lists as Objects .....	298
<b>HOUR 20: Using Special Classes, Functions, and Pointers</b>	<b>301</b>
Static Member Data .....	301
Static Member Functions .....	303
Containment of Classes .....	305
Friend Classes and Functions .....	312
<b>HOUR 21: Using New Features of C++14</b>	<b>331</b>
The Newest Version of C++ .....	331
Using <code>auto</code> in Function Return Types .....	332
Improved Numeric Literals .....	334
The <code>constexpr</code> Keyword .....	335
Lambda Expressions .....	338
<b>HOUR 22: Employing Object-Oriented Analysis and Design</b>	<b>343</b>
The Development Cycle .....	343
Simulating an Alarm System .....	344
PostMaster: A Case Study .....	350
<b>HOUR 23: Creating Templates</b>	<b>373</b>
What Are Templates? .....	373
Instances of the Template .....	374
Template Definition .....	374
Using Template Items .....	381
<b>HOUR 24: Dealing with Exceptions and Error Handling</b>	<b>391</b>
Bugs, Errors, Mistakes, and Code Rot .....	391
Handling the Unexpected .....	392
Exceptions .....	393
Using <code>try</code> and <code>catch</code> Blocks .....	397
Writing Professional-Quality Code .....	403

**Part VII: Appendixes**

<b>APPENDIX A: Binary and Hexadecimal</b>	<b>411</b>
Other Bases .....	412
Around the Bases .....	412
Hexadecimal .....	416
<b>APPENDIX B: Glossary</b>	<b>419</b>
<b>APPENDIX C: This Book's Website</b>	<b>427</b>
<b>APPENDIX D: Using the MinGW C++ Compiler on Windows</b>	<b>429</b>
Downloading MinGW-w64 .....	429
Setting the Path Environment Variable .....	431
Testing Your Installation .....	433
<b>Index</b>	<b>439</b>

# About the Authors

**Rogers Cadenhead** is a writer, computer programmer, and web developer who has written more than 25 books on Internet-related topics, including *Sams Teach Yourself Java in 21 Days* and *Absolute Beginner's Guide to Minecraft Mods Programming*. He publishes the Drudge Retort and other websites that receive more than 22 million visits a year. This book's official website is at <http://cplusplus.cadenhead.org>.

**Jesse Liberty** is the author of numerous books on software development, including best-selling titles on C++ and .NET. He is the president of Liberty Associates, Inc. ([www.libertyassociates.com](http://www.libertyassociates.com)), where he provides custom programming, consulting, and training.

# 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](mailto:feedback@sampublishing.com)

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

## Reader Services

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

# Introduction

Congratulations! By reading this sentence, you are already 20 seconds closer to learning C++, one of the most important programming languages in the world.

If you continue for another 23 hours, 59 minutes, and 40 seconds, you will master the fundamentals of the C++ programming language. Twenty-four one-hour lessons cover important features such as managing I/O, creating loops and arrays, using object-oriented programming with templates, and creating C++ programs.

All of this has been organized into well-structured, easy-to-follow lessons. There are working projects that you create—complete with output and an analysis of the code—to illustrate the topics of the hour. Syntax examples are clearly marked for handy reference.

To help you become more proficient, each hour ends with a set of common questions and answers.

## Who Should Read This Book?

You don't need any previous experience in programming to learn C++ with this book. It starts with the basics and teaches you both the language and the concepts involved with programming C++. Whether you are just beginning or already have some experience programming, you will find that this book makes learning C++ fast and easy.

## Should I Learn C First?

No, you don't need to learn C first. C++ is a much more powerful and versatile language that was created by Bjarne Stroustrup as a successor to C. Learning C first can lead you into some programming habits that are more error-prone than what you'll do in C++. This book does not assume that readers are familiar with C.

## Why Should I Learn C++?

You could be learning a lot of other languages, but C++ is valuable to learn because it has stood the test of time and continues to be a popular choice for modern programming.

Despite being created in 1979, C++ is still being used for professional software today because of the power and flexibility of the language. There's even a new version of the language, called C++14, that makes it even more useful.

Because other languages such as Java were inspired by C++, learning the language can provide you insight into them, as well. Mastering C++ gives you portable skills that you can use on just about any platform on the market today, from desktop computers to Linux servers, mobile devices, videogame consoles, and mainframes.

## What If I Don't Want This Book?

I'm sorry you feel that way, but these things happen sometimes. Please reshelve this book with the front cover facing outward on an endcap with access to a lot of the store's foot traffic.

## Conventions Used in This Book

This book contains special elements as described here.

### NOTE

---

These boxes provide additional information to the material you just read.

---

### CAUTION

---

These boxes focus your attention on problems or side effects that can occur in specific situations.

---

### TIP

---

These boxes give you tips and highlight information that can make your C++ programming more efficient and effective.

---

When you see this symbol, you know that what you see next will show the output from a code listing/example.



This book uses various typefaces:

- ▶ To help you distinguish C++ code from regular English, actual C++ code is typeset in a special `monospace` font.
- ▶ Placeholders—words or characters temporarily used to represent the real words or characters you would type in code—are typeset in *italic monospace*.
- ▶ New or important terms are typeset in *italic*.
- ▶ In the listings in this book, each real code line is numbered. If you see an unnumbered line in a listing, you'll know that the unnumbered line is really a continuation of the preceding numbered code line (some code lines are too long for the width of the book). In this case, you should type the two lines as one; do not divide them.

*This page intentionally left blank*

# HOUR 2

## Organizing the Parts of a Program

---

### What You'll Learn in This Hour:

- ▶ Why to use C++
- ▶ How C++ programs are organized
- ▶ How comments make programs easier to understand
- ▶ What functions can accomplish

Although it recently turned 37, the C++ programming language has aged a lot better than some other things that came out in the late 1970s. Unlike disco, oil embargoes, shag carpet, and avocado-colored refrigerators, C++ is still in vogue today. It remains a world-class programming language.

The reason for its surprising longevity is that C++ makes it possible to create fast executing programs with a small amount of code that can run on a variety of computing environments. Today's C++ programming tools enable the creation of complex and powerful applications in commercial, business, and open source development.

### Reasons to Use C++

During the seven decades of the computing age, computer programming languages have undergone a dramatic evolution. C++ is considered to be an evolutionary improvement of a language called C that was introduced in 1972.

The earliest programmers worked with the most primitive computer instructions: machine language. These instructions were represented by long strings of ones and zeroes. Assemblers were devised that could map machine instructions to human-readable and manageable commands such as `ADD` and `MOV`.

The instructions that make up a computer program are called its *source code*.

In time, higher-level languages were introduced such as BASIC and COBOL. These languages made it possible for programmers to begin to craft programs using language closer to actual

words and sentences, such as `Let Average = .366`. These instructions were translated back into machine language by tools that were called either *interpreters* or *compilers*.

An interpreter-based language translates a program as it reads each line, acting on each instruction.

A compiler-based language translates a program into what is called *object code* through a process called *compiling*. This code is stored in an object file. Next, a linker transforms the object file into an executable program that can be run on an operating system.

Because interpreters read the code as it is written and execute the code on the fly, they're easy for programmers to work with. Compilers require the more inconvenient extra steps of compiling and linking programs. The benefit to this approach is that the programs run significantly faster than programs run by an interpreter.

For many years, the principal goal of programmers was to write short pieces of code that would execute quickly. Programs needed to be small because memory was expensive, and they needed to be fast because processing power also was expensive. As computers have become cheaper, faster, and more powerful and the cost and capacity of memory has fallen, these priorities diminished in importance.

Today, the greatest expense in programming is the cost of a programmer's time. Modern languages such as C++ make it faster to produce well-written, easy-to-maintain programs that can be extended and enhanced.

## **Styles of Programming**

As programming languages have evolved, languages have been created to cater to different styles of programming.

In procedural programming, programs are conceived of as a series of actions performed on a set of data. Structured programming was introduced to provide a systematic approach to organizing these procedures and managing large amounts of data.

The principal idea behind structured programming is to divide and conquer. Take a task that needs to be accomplished in a program, and if it is too complex, break it down into a set of smaller component tasks. If any of those tasks is still too complicated, break it down into even smaller tasks. The end goal is tasks that are small and self-contained enough to be easily understood.

As an example, pretend you've been asked by this publisher to write a program that tracks the average income of its team of enormously talented and understatedly charismatic computer book authors. This job can be broken down into these subtasks:

- 1.** Find out what each author earns.
- 2.** Count how many authors the publisher has.

3. Total all their income.
4. Divide the total by the number of authors.

Totaling the income can be broken down into the following:

1. Get each author's personnel record.
2. Access the author's book advances and royalties.
3. Deduct the cost of morning coffee, corrective eyewear and chiropractic care.
4. Add the income to the running total.
5. Get the next author's record.

In turn, obtaining each author's record can be broken down into these subtasks:

1. Open the file folder of authors.
2. Go to the correct record.
3. Read the data from disk.

Although structured programming has been widely used, this approach has some drawbacks. The separation of data from the tasks that manipulate the data becomes harder to comprehend and maintain as the amount of data grows. The more things that must be done with data, the more confusing a program becomes.

Procedural programmers often find themselves reinventing new solutions to old problems instead of producing reusable programs. The idea behind reusability is to build program components that can be plugged into programs as needed. This approach is modeled after the physical world, where devices are built out of individual parts that each perform a specific task and have already been manufactured. A person designing a bicycle doesn't have to create a brake system from scratch. Instead, she can incorporate an existing brake into the design and take advantage of its existing functionality.

This component-based approach became available to computer programmers for the first time with the introduction of object-oriented programming.

## **C++ and Object-Oriented Programming**

C++ helped popularize a revolutionary style of programming with a funny acronym: OOP.

The essence of object-oriented programming is to treat data and the procedures that act upon the data as a single object—a self-contained entity with an identity and characteristics of its own.

The C++ language fully supports object-oriented programming, including three concepts that have come to be known as the pillars of object-oriented development: encapsulation, inheritance, and polymorphism.

## **Encapsulation**

When the aforementioned bike engineer creates a new bicycle, she connects together component pieces such as the frame, handlebars, wheels, and a headlight (baseball card in the spokes optional). Each component has certain properties and can accomplish certain behaviors. She can use the headlight without understanding the details of how it works, as long as she knows what it does.

To achieve this, the headlight must be self-contained. It must do one well-defined thing and it must do it completely. Accomplishing one thing completely is called *encapsulation*.

All the properties of the headlight are encapsulated in the headlight object. They are not spread out through the bicycle.

C++ supports the properties of encapsulation through the creation of user-defined types called *classes*. A well-defined class acts as a fully encapsulated entity that is used as an entire unit or not at all. The inner workings of the class should be hidden on the principle that the programs which use a well-defined class do not need to know how the class works. They only need to know is how to use it. You learn how to create classes in Hour 8, “Creating Basic Classes.”

## **Inheritance and Reuse**

Now we’re starting to learn a little more about our bike engineer. Let’s call her Penny Farthing. Penny needs her new bicycle to hit the market quickly—she has run up enormous gambling debts to people who are not known for their patience.

Because of the urgency, Penny starts with the design of an existing bicycle and enhances it with cool add-ons like a cup holder and mileage counter. Her new bicycle is conceived as a kind of bicycle with added features. She reused all the features of a regular bicycle while adding capabilities to extend its utility.

C++ supports the idea of reuse through inheritance. A new type can be declared that is an extension of an existing type. This new subclass is said to derive from the existing type. Penny’s bicycle is derived from a plain old bicycle and thus inherits all its qualities but adds additional features as needed. Inheritance and its application in C++ are discussed in Hour 16, “Extending Classes with Inheritance.”

## **Polymorphism**

As its final new selling point, Penny Farthing’s Amazo-Bicycle behaves differently when its horn is squeezed. Instead of honking like an anguished duck, it sounds like a car when lightly pressed

and roars like a foghorn when strongly squashed. The horn does the right thing and makes the proper sound based on how it is used by the bicycle's rider.

C++ supports this idea that different objects do the right thing through a language feature called *function polymorphism* and *class polymorphism*. *Polymorphism* refers to the same thing taking many forms, and is discussed during Hour 17, "Using Polymorphism and Derived Classes."

You will learn the full scope of object-oriented programming by learning C++. These concepts will become familiar to you by the time you've completed this full 24-hour ride and begun to develop your own C++ programs.

Disclaimer: You won't learn how to design bicycles or get out of gambling debt.

## The Parts of a Program

The program you created during the first hour, `Motto.cpp`, contains the basic framework of a C++ program. Listing 2.1 reproduces the source code of this program so that it can be explored in more detail.

When typing this program in to a programming editor such as NetBeans, remember not to include the line numbers in the listing. They are included solely for the purpose of referring to specific lines in this book.

### LISTING 2.1 The Full Text of `Motto.cpp`

---

```

1: #include <iostream>
2:
3: int main()
4: {
5:     std::cout << "Solidum petit in profundis!\n";
6:     return 0;
7: }
```

---

This program produces a single line of output, the motto of Aarhus University:

```
Solidum petit in profundis!
```

On line 1 of Listing 2.1 a file named `iostream` is included in the source code. This line causes the compiler to act as if the entire contents of that file were typed at that place in `Motto.cpp`.

## Preprocessor Directives

A C++ compiler's first action is to call another tool called the preprocessor that examines the source code. This happens automatically each time the compiler runs.

The first character in line 1 is the # symbol, which indicates that the line is a command to be handled by the preprocessor. These commands are called *preprocessor directives*. The preprocessor's job is to read source code looking for directives and modify the code according to the indicated directive. The modified code is fed to the compiler.

The preprocessor serves as an editor of code right before it is compiled. Each directive is a command telling that editor what to do.

The `#include` directive tells the preprocessor to include the entire contents of a designated filename at that spot in a program. As you learned in Hour 1, "Writing Your First Program," C++ includes a standard library of source code that can be used in your programs to perform useful functionality. The code in the `iostream` file supports input and output tasks such as displaying information onscreen and taking input from a user.

The `<` and `>` brackets around the filename `iostream` tell the preprocessor to look in a standard set of locations for the file. Because of the brackets, the preprocessor looks for the `iostream` file in the folder that holds header files for the compiler. These files also are called *include files* because they are included in a program's source code.

The full contents of `iostream` are included in place of line 1.

#### NOTE

---

Header files traditionally ended with the filename extension `.h` and also were called *h files*, so they used a directive of the form `include <iostream.h>`.

Modern compilers don't require that extension, but if you refer to files using it, the directive might still work for compatibility reasons. This book omits the extraneous `.h` in include files.

---

The contents of the file `iostream` are used by the `cout` command in line 5, which displays information to the screen.

There are no other directives in the source code, so the compiler handles the rest of `Motto.cpp`.

## Source Code Line by Line

Line 3 begins the actual program by declaring a function named `main()`. *Functions* are blocks of code that perform one or more related actions. Functions do some work and then return to the spot in the program where they were called.

Every C++ program has a `main()` function. When a program starts, `main()` is called automatically.

All functions in C++ must return a value of some kind after their work is done. The `main()` function always returns an integer value. Integers are specified using the keyword `int`.



Functions, like other blocks of code in a C++ program, are grouped together using the brace marks { and }. All functions begin with an opening brace { and end with a closing brace }.

The braces for the `main()` function of `Motto.cpp` are on lines 4 and 7, respectively. Everything between the opening and closing braces is part of the function.

In line 5, the `cout` command is used to display a message on the screen. The object has the designation `std::` in front of it, which tells the compiler to use the standard C++ input/output library. The details of how this works are too complex for this early hour and likely will cause you to throw the book across the room if introduced here. For the safety of others in your vicinity, they are explained later. For now, treat `std::cout` as the name of the object that handles output in your programs and `std::cin` as the object that handles user input.

The reference to `std::cout` in line 5 is followed by `<<`, which is called the *output redirection operator*. Operators are characters in lines of code that perform an action in response to some kind of information. The `<<` operator displays the information that follows it on the line. In line 5, the text `"Solidum petit in profundis!\n"` is enclosed within double quotes. This displays a string of characters on the screen followed by a special character specified by `"\n"`, a newline character that advances the program's output to the beginning of the next line.

On line 6, the program returns the integer value 0. This value is received by the operating system after the program finishes running. Typically, a program returns the value 0 to indicate that it ran successfully. Any other number indicates a failure of some kind.

The closing braces on line 7 ends the `main()` function, which ends the program. All of your programs use the basic framework demonstrated by this program.

## Comments

As you are writing your own programs for the first time, it will seem perfectly clear to you what each line of the source code does. But as time passes and you come back to the program to fix a bug or add a new feature, you may find yourself completely mystified by your own work.

To avoid this predicament and help others understand your program, you can document your source code with comments. *Comments* are lines of text that explain what a program is doing. The compiler ignores them, so they are strictly for benefit of humans reading the code.

There are two types of comments in C++. A single-line comment begins with two slash marks (`//`) and causes the compiler to ignore everything that follows the slashes on the same line. Here's an example:

```
// The next line is a kludge (ugh!)
```

A multiple-line comment begins with the slash and asterisk characters (`/*`) and ends with the same characters reversed (`*/`). Everything within the opening `/*` and the closing `*/` is a comment, even if it stretches over multiple lines. If a program contains a `/*` that is not followed by a `*/` somewhere, that's an error likely to be flagged by the compiler. Here's a multiline comment:

```
/* This part of the program doesn't work very well. Please remember to
   fix this before the code goes live -- or else find a scapegoat you can
   blame for the problem. The new guy Curtis would be a good choice. */
```

In the preceding comment, the text on the left margin is lined up to make it more readable. This is not required. Because the compiler ignores everything within the `/*` and `*/`, anything can be put there—grocery lists, love poems, secrets you've never told anybody in your life, and so on.

### CAUTION

---

An important thing to remember about multiline comments is that they do not nest inside each other. If you use one `/*` to start a comment and then use another `/*` a few lines later, the first `*/` mark encountered by the compiler will end all multiline comments. The second `*/` mark will result in a compiler error. Most C++ programming editors display comments in a different color to make clear where they begin and end.

---

The next project that you create includes both kinds of comments. Write lots of comments in your programs. The more time spent writing comments that explain what's going on in source code, the easier that code will be to work on weeks, months or even years later.

## Functions

The `main()` function is unusual among C++ functions because it's called automatically when a program begins running.

A program is executed line by line in source code, beginning with the start of `main()`. When a function is called, the program branches off to execute the function. After the function has done its work, it returns control to the line where the function was called. Functions may or may not return a value, with the exception of `main()`, which always returns an integer.

Functions consist of a header and a body. The header consists of three things:

- ▶ The type of data the function returns
- ▶ The function's name
- ▶ The parameters received by the function

The *function name* is a short identifier that describes its purpose.

When a function does not return a value, it uses data type `void`, which means nothing. To clarify: `void` isn't meaningless. It means "nothing," like how stars in space are separated by a ginormous amount of nothing called "the void."

*Arguments* are data sent to the function that control what it does. These arguments are received by the function as *parameters*. A function can have zero, one, or more parameters. The next program that you create has a function called `add()` that adds two numbers together. Here's how it is declared:

```
int add(int x, int y)
{
    // body of function goes here
}
```

The parameters are organized within parentheses marks as a list separated by commas. In this function, the parameters are integers named `x` and `y`.

The name of a function, its parameters and the order of those parameters is called its *signature*. Like a person's signature, the function's signature uniquely identifies it.

A function with no parameters has an empty set of parentheses, as in this example:

```
int getServerStatus()
{
    // body of function here
}
```

Function names cannot contain spaces, so the `getServerStatus()` function capitalizes the first letter of each word after the first one. This naming convention is common among C++ programmers and adopted throughout this book.

The body of a function consists of an opening brace, zero or more statements, and a closing brace. A function that returns a value uses a `return` statement, as you've seen in the `Motto` program:

```
return 0;
```

The `return` statement causes a function to exit. If you don't include at least one `return` statement in a function, it automatically returns `void` at the end of the function's body. In that situation, `void` must be specified as the function's return type.

## Using Arguments with Functions

The `Calculator.cpp` program in Listing 2.2 fleshes out the aforementioned `add()` function, using it to add a pair of numbers together and display the results. This program demonstrates how to create a function that takes two integer arguments and returns an integer value.

**LISTING 2.2** The Full Text of `Calculator.cpp`

---

```
1: #include <iostream>
2:
3: int add(int x, int y)
4: {
5:     // add the numbers x and y together and return the sum
6:     std::cout << "Running calculator ...\\n";
7:     return (x+y);
8: }
9:
10: int main()
11: {
12:     /* this program calls an add() function to add two different
13:        sets of numbers together and display the results. The
14:        add() function doesn't do anything unless it is called by
15:        a line in the main() function. */
16:     std::cout << "What is 867 + 5309?\\n";
17:     std::cout << "The sum is " << add(867, 5309) << "\\n\\n";
18:     std::cout << "What is 777 + 9311?\\n";
19:     std::cout << "The sum is " << add(777, 9311) << "\\n";
20:     return 0;
21: }
```

---

This program produces the following output:

```
What is 867 + 5309?
Running calculator ...
The sum is 6176
```

```
What is 777 + 9311?
Running calculator ...
The sum is 10088
```

The `Calculator` program includes a single-line comment on line 5 and a multi-line comment on lines 12–15. All comments are ignored by the compiler.

The `add()` function takes two integer parameters named `x` and `y` and adds them together in a return statement (lines 3–8).

The program's execution begins in the `main()` function. The first statement in line 16 uses the object `std::cout` and the redirection operator `<<` to display the text "What is 867 + 5309?" followed by a newline.

The next line displays the text "The sum is" and calls the `add()` function with the arguments 777 and 9311. The execution of the program branches off to the `add()` function, as you can tell in the output by the text "Running calculator...."

The integer value returned by the function is displayed along with two more newlines.

The process repeats for a different set of numbers in lines 18–19.

The formula  $(x+y)$  is an expression. You learn how to create these mathematical workhorses in Hour 4, “Using Expressions, Statements, and Operators.”

## Summary

During this hour, you were shown how C++ evolved from other styles of computer languages and embraced a methodology called object-oriented programming. This methodology has been so successful in the world of computing that the language remains as contemporary today as it did when it was invented in 1979.

I wish the mullet haircut I sported in college had survived the test of time as well. Instead, it lives on in Facebook photos that friends share to shock and awe.

In the two programs that you developed during this hour, you made use of three parts of a C++ program: preprocessor directives, comments, and functions.

All the programs that you will create in C++ employ the same basic framework as the Motto and Calculator programs. They just become more sophisticated as they make use of more functions, whether you write them from scratch or call functions from header files included with the `#include` directive.

## Q&A

**Q. What does the # character do in a C++ program?**

**A.** The # symbol signals that the line is a preprocessor directive, a command that is handled before the program is compiled. The `#include` directive includes the full text of a file at that position in the program. The compiler never sees the directive. Instead, it acts as if the contents of the file were typed in with the rest of the source code.

**Q. What is the difference between // comments and /\* style comments?**

**A.** The comments that start with // are single-line comments that end with the end of the line on which they appear. The /\* comments are multi-line comments that don't end until a \*/ is encountered. The end of a function won't even cause a multi-line comment to be ended. You must put in the closing \*/ mark or the compiler will fail with an error.

**Q. What's the difference between function arguments and function parameters?**

**A.** The terms are two sides of the same process when a function is called. Arguments are the information sent to the function. Parameters are the same information received by the function. You call a function with arguments. Within a function, those arguments are received as parameters.

**Q. What is a kludge?**

- A.** A kludge is an ugly solution to a problem that's intended to be replaced later with something better. The term was popularized by Navy technicians, computer programmers, and aerospace engineers and spread to other technical professions.

In a computer program a kludge is source code that works but would have been designed better if there had been more time. Kludges have a tendency to stick around a lot longer than expected.

The astronauts on the Apollo 13 mission created one of the greatest kludges of all time: a system cobbled together from duct tape and socks that filtered carbon dioxide from the air on the spacecraft and helped them make it back to Earth.

The first known usage of the term was in a 1962 article in *Datamation* magazine by Jackson W. Granholm, who gave it an elegant definition that has stood the test of time: "An ill-assorted collection of poorly matching parts, forming a distressing whole."

## Workshop

Now that you've learned about some of the pieces of a C++ program, you can answer a couple of questions and complete a couple of exercises to firm up your knowledge.

## Quiz

1. What data type does the `main` function return?
  - A. `void`
  - B. `int`
  - C. It does not return a type.
2. What do the braces do in a C++ program?
  - A. Indicate the start and end of a function
  - B. Indicate the start and end of a program
  - C. Straighten the program's teeth
3. What is not part of a function's signature?
  - A. Its name
  - B. Its arguments
  - C. Its return type

## Answers

- 1. B.** The `main` function returns an `int` (integer).
- 2. A.** Braces mark the start and end of functions and other blocks of code you learn about in upcoming hours.
- 3. C.** A function signature consists of its name, parameters, and the precise order of those parameters. It does not include its return type.

## Activities

- 1.** Rewrite the `Motto` program to display the Aarhus University motto in a function.
- 2.** Rewrite the `Calculator` program to add a third integer called `z` in the `add()` function and call this function with two sets of three numbers.

To see solutions to these activities, visit this book's website at <http://cplusplus.cadenhead.org>.

*This page intentionally left blank*



# Index

## Symbols

- +** (addition) operator
  - explained, 46
  - overloading, 219–220
- &** (address-of) operator, 169–171
- &&** (AND) operator, 56–57
- =** (assignment) operator
  - compared to copy constructor, 228
  - explained, 33, 45–46
  - overloading, 221–223
- \*** (asterisk) with pointers, 169
- \** (backslash), 137
- { }** (braces), 19, 62, 403–404
- <>** brackets, 18
- ,** (comma), 335
- //** comment notation, 19, 23
- /\* \*/** comment notation, 20, 23
- (decrement) operator, 47
- /** (division) operator, 46
- .** (dot) operator, 103, 112, 157
- "** (double quotes), 137
- ==** (equality) operator
  - explained, 52
  - overloading, 224
- !** (exclamation point)
  - ! (NOT) operator, 57
  - != (inequality) operator, 52
  - numeric value, 31
- >** (greater than) operator, 52
- >=** (greater than or equals) operator, 52
- #** (hash) symbol, 23
- ++** (increment) operator
  - explained, 47
  - overloading
    - postfix operator, 217–218
    - prefix operator, 215–216
- \*** (indirection) operator, 140–141, 169
- <** (less than) operator, 52
- <=** (less than or equals) operator, 52
- %** (modulus) operator, 46
- \*** (multiplication) operator, 46, 169
- ||** (OR) operator, 57
- r**, 19
- &** (reference) operator, 169
- +=** (self-assigned addition) operator, 46–47
- %=** (self-assigned modulus) operator, 46–47

**\*= (self-assigned multiplication) operator, 46–47**

**-= (self-assigned subtraction) operator, 46–47**

**' (single quotes)**  
 digit separator, 335  
 displaying, 137

**– (subtraction operator), 46**

**:: (scope resolution operator), 114**

**;(semicolon)**  
 common mistakes, 86  
 in compound statements, 44  
 with variables, 32

**~ (tilde), 117, 203**

**\_ (underscore), 32**

**13, superstitions related to, 107**

**80/20 rule, 358**

**80/80 rule, 358**

## A

**Aarhus University, 8**

**abort() function, 399**

**abstract data types (ADT)**  
 advantages of, 284  
 complex hierarchies of, 279–283  
 explained, 271–275, 346  
 when to use, 283–284

**access guidelines, 407**

**accessing**  
 classes  
 class members, 112–113  
 contained classes, 311  
 data members, 157–158

**accessors, 113, 123**

**add() function, 21–23**

**addition operator (+)**  
 explained, 46  
 overloading, 219–220

**Addresser program, 136–137**

**addresses**  
 examining in pointers, 143–144  
 storing in pointers, 138–140

**address-of operator (&), 169–171**

**ADTs. See abstract data types (ADT)**

**advantages of C++, 13–14**

**airplane black boxes, 92**

**Alabama Crimson Tide, 409**

**allocating**  
 linked lists, 294  
 memory, 146–147

**analysis and requirements, 345**

**angle brackets (<>), 18**

**Animal program, 280–283**

**anonymous functions (inline), 338–339**

**APIs (application programming interfaces), 359–360**

**apostrophe (')**  
 digit separator, 335  
 displaying, 137

**application programming interfaces (APIs), 359–360**

**architecture prototypes, 358**

**Area program, 63**

**AreaCube program, 71**

**arguments**  
 compared to parameters, 23  
 explained, 21–23

passing by reference, 172–174  
 passing to base constructors, 239–244

**ArrayFunction program, 316–318**

**ArrayMaxer program, 337**

### arrays

arrays of pointers  
 to functions, 316–318  
 to member functions, 325–327

buffer overflow, 107

character arrays, 102–104

constant expressions, 335–338

declaring, 95

elements of, 95

explained, 95–97

initializing, 98

memory, 101–102

multidimensional arrays  
 explained, 99–100  
 initializing, 100–101

reading with foreach loops, 105–106

sizeof() function, 337–338

uninitialized array elements, 107

writing past end of, 97–98

**Assignment class, 222–223**

**assignment operator (=)**

compared to copy constructor, 228  
 explained, 33, 45–46  
 overloading, 221–223

**Assignment program, 170–171**

asterisk (\*), 169

auto keyword

- explained, 38, 62
- in function return types, 332–334
- risks, 339

AutoArea program, 333

AutoCube program, 74–75

auto-typed return values, 73–75

auto-typed variables, 38–40

Autri, Gene, 265

avoiding memory leaks, 149

## B

backslash, displaying, 137

Badger program, 84

BadTeacher program, 90–91

base classes

- initializing, 239–244
- member functions
  - calling, 247–249
  - constructors, passing arguments to, 239–244
  - hiding, 246–247

base constructors, passing arguments to, 239–244

BASIC, 13–14

Bauman, Charlie, 328

Beaudouin, Paul, 92

Bell Labs, 5–6

binary integers, 334–335

binary literals, 335

binding, late, 255

black boxes (airplane), 92

bool type, 28, 31

Boole, George, 28

Boolean variables, 28

Box program, 100–101

BoxMaker program, 87–88

braces ({}), 19, 62, 403–404

break statement, 81–82

breaking out of loops, 81–82

BridgeKeeper program, 103–104

buffer overflow, 107

buffers, 102

bug-free code. *See* professional-quality code, writing

bugs. *See* errors

bulletproof programs, 392

## C

.c file extension, 7

C language, 5–6

*The C++ Programming Language* (Stroustrup), 331

C++ standardization website, 340

C++0x, 331

C++1z, 340

C++14, 331, 332, 333

Calculator program, 21–23

calling base class member functions, 247–249

CamelCase, 33, 406

Canary Islands, Silbo language in, 284–285

capitalization of names, 406

case sensitivity, 32, 112

casting

- dynamic casting, 284
- explicit casts, 257

catch blocks, 393–397

catching exceptions

- advantages of, 408
- definition of, 393
- explained, 397–398
- multiple catches, 398
- polymorphism, 398–403
- by reference, 398–403
- try/catch blocks, 393–397

char type, 28, 31, 102

character arrays, 102–104

Circle program, 209

classes. *See also* objects; polymorphism

- accessing members of, 112–113
- Assignment, 222–223
- base classes
  - base class member functions, 246–249
  - constructors, 239–244
  - initializing, 239–244
- class definitions
  - organizing, 124
  - writing, 407
- contained classes
  - accessing members of, 311
  - copying by value versus copying by reference, 312
  - defining, 305–311
  - filtering access to, 312
- declaring, 111

- definition of, 7, 16
- design
  - overview, 346–348
  - PostMaster case study, 353–354
- encapsulation, 110
- explained, 110–111
- friend classes
  - disadvantages of, 327
  - explained, 312–313
- HeadNode, 290–297
- inheritance
  - base class initialization, 239–244
  - calling base class member functions, 247–249
  - constructors, 236–239
  - derivation, 232
  - destructors, 236–239
  - explained, 16, 231
  - hiding base class member functions, 246–247
  - overriding functions, 244–246
  - protected classes, 234–236
  - syntax, 233–234
- InternalNode, 290–297
- member functions. *See* functions
- member variables, 110
- Number, 213
- object initialization, 203–204
- pAddress, 368–369
- pObject, 357
- Point, 127–128
- PostMasterMessage, 358–359
- private classes, 112–113, 234
- protected classes, 234–236
- public class members, 112–113, 121
- Rectangle, 127–128
- String, 306–309
- TailNode, 290–297
- Tricycle, 111, 125–126, 205–207
- clauses, *else*, 53–54**
- clone() function, 260–263**
- clown colleges, 196**
- COBOL, 13–14**
- code listings. *See* programs**
- code space, 145**
- colon (:), 114**
- Combat program, 39**
- combining operators, 46–47**
- comma (,), 335**
- comments, 19–20, 23, 406–407**
- Compass program, 37–38**
- compilers**
  - constructors provided by compiler, 118–120
  - explained, 5–6
  - GCC, 7
  - history of, 13–14
  - type inference, 332–334
  - warning messages from, 9
- compile-time constant expressions, 208–210**
- compiling**
  - definition of, 14
  - explained, 5–7
- components of linked lists, 289–297**
- compound statements**
  - compound if, 54–56
  - explained, 44
- conceptualization, 344**
- const keyword, 123, 132, 208, 247, 408**
- const member functions**
  - classes with other classes as member data, 127–132
  - declaring, 123
  - explained, 123, 124
  - inline implementation, 124–127
  - interface versus implementation, 124
- const objects, 165**
- const pointers**
  - explained, 162–164
  - passing, 186–188
- constant expressions, 335–338**
- Constantinople, 249**
- constants**
  - compile-time constant expressions, 208–210
  - defining, 36–37
  - enumerated constants, 37–38
  - explained, 36
  - literal constants, 36
  - null pointer constant, 150–151
  - symbolic constants, 36, 41
- constexpr keyword, 209, 335–338**
- ConstPasser program, 186–188**
- ConstPointer program, 163–164**
- constructors**
  - base constructors, passing arguments to, 239–244

- constructors provided by
    - compiler, 118–120
  - copy constructors, 228
  - declaring, 116–117, 210–211
  - default constructors, 112–117
  - definition of, 117
  - explained, 204–208
  - inheritance, 236–239
  - overloading, 203
  - contained classes**
    - accessing members of, 311
    - copying by value versus copying by reference, 312
    - defining, 305–311
    - filtering access to, 312
  - continue statement, 82–83**
  - continuing to next loop, 82–83**
  - conversion operators, 224–226**
  - convert() function, 62, 65**
  - copy constructors**
    - explained, 183, 204–208, 228
    - virtual copy constructors, 260–263
  - copying**
    - deep copying, 204–208
    - shallow copying, 204
    - strings, 104–105
  - Coulier, Dave, 228**
  - Counter program, 214**
  - Counter2 program, 215–216**
  - Counter3 program, 217–218**
  - Counter4 program, 219–220**
  - Counter5 program, 224**
  - Counter6 program, 225**
  - Counter7 program, 226–227**
  - .cp file extension, 7**
  - .cpp file extension, 7**
  - Crimson Tide, 409**
  - crossword puzzles, invention of, 340**
  - .cxx file extension, 7**
- D**
- dangling pointers, 161–162**
  - data members**
    - accessing with pointers, 157–158
    - definition of, 111
  - data types. See types**
  - DataMember program, 158–159**
  - decimal values, assigning to integers, 41**
  - declaring. See also defining**
    - anonymous functions (inline), 338–339
    - arrays, 95
    - classes, 111
    - constructors, 116–117, 210–211
    - destructors, 117
    - functions
      - const member functions, 123
      - example, 61–64
    - pointers
      - to functions, 313–316
      - to variables, 138
    - pure virtual functions, 275–276
    - references, 167–168
    - virtual function members, 252–253
  - decrement operator (–), 47**
  - deep copying, 204–208**
  - DeepCopy program, 205–207**
  - default constructors, 112–117**
  - default function parameters, 70–72**
  - default values**
    - advantages/ disadvantages, 210
    - member functions, 201–203
  - #define directive, 36**
  - defining. See also declaring**
    - constants, 36–37
    - contained classes, 305–311
    - functions, 61–64
    - objects, 111–112, 116–117
    - static member variables, 303
    - templates, 374–381
    - types, 110
    - variables, 32–33
  - delegation of responsibility, 289**
  - delete keyword, 147–149**
  - deleting objects, 116–117, 155–157**
  - derivation, inheritance and, 232**
  - design**
    - analysis and requirements, 345
    - classes, 346–348
    - conceptualization, 344
    - development cycle, 343
    - event loops, 348–350
    - high-level design, 345–346
    - low-level design, 345–346
    - PostMaster case study
      - 80/20 rule, 358
      - 80/80 rule, 358

- API (application programming interface), 359–360
  - driver programs, 362–369
  - initial class design, 353–354
  - interfaces, 356–357
  - message format, 352–353
  - ongoing design considerations, 361–362
  - overview, 350–351
  - PostMasterMessage class, 358–359
  - programming in large groups, 360
  - prototype, 357–358
  - rooted versus non-rooted hierarchies, 354–355
  - subprojects, 351–352
  - simulation, 344
  - third-party libraries, 356
  - destructors**
    - explained, 117
    - inheritance, 236–239
    - overloading, 203
    - virtual destructors, 260
  - development cycle, 343**
  - development of C++, 5–6, 331**
  - Diogenes, 299**
  - directives**
    - #define, 36
    - #include, 104, 309
    - overview, 17–18
  - displaying**
    - backslash, 137
    - quotation marks, 137
  - division operator (/), 46**
  - dot operator (.), 112, 157**
  - double quotes ("), 137**
  - double type, 31**
  - doubly linked lists, 287–288**
  - do-while loops, 83–84**
  - draw() function, 276, 279**
  - drawShape() function, 201**
  - Driver program, 362–368**
  - driver programs, 362–369**
  - Dusky Seaside Sparrow, 181**
  - dynamic casting, 284**
- E**
- easy-to-read code, writing, 405**
  - Edit() function, 369**
  - editors. See text editors**
  - elements of arrays, 95, 107**
  - elephant mascot (Alabama Crimson Tide), 409**
  - else clause, 53–54**
  - Emerson, Ralph Waldo, 403**
  - Employee program, 309–311**
  - encapsulation**
    - definition of, 110
    - explained, 16
  - enum keyword, 37**
  - enumerated constants, 37–38**
  - enumerations, 37**
  - equality operator (==)**
    - explained, 52
    - overloading, 223
  - errors. See also exceptions**
    - buffer overflow, 107
    - fence post errors, 97
  - overview, 391–392
  - professional-quality code, writing
    - access, 407
    - braces ({ }), 403–404
    - class definitions, 407
    - comments, 406–407
    - const, 408
    - identifier names, 405–406
    - include files, 408
    - long lines, 404
    - overview, 403
    - program text, 405
    - spelling and capitalization of names, 406
    - switch statements, 404–405
  - event loops, 348–350**
  - examining addresses stored in pointers, 143–144**
  - Exception program, 394–397**
  - exceptions**
    - catching
      - advantages of, 408
      - definition of, 393
      - explained, 397–398
      - multiple catches, 398
      - polymorphism, 398–403
      - by reference, 398–403
      - try/catch blocks, 393–397
    - definition of, 392–393
    - explained, 393
    - throwing, 393
    - try/catch blocks, 393–397
  - exclamation point (!), 31**
  - explicit casts, 257**
  - Expression program, 45**

**expressions**

- compile-time constant expressions, 208–210
- constant expressions, 335–338
- explained, 44–45
- expression values, 58
- lambda expressions, 338–339

**F****factor() function, 179, 180****Favre, Henri, 370****Fell, Norman, 41****fence post errors, 97****Fifteens program, 82–83****files. See also programs**

- header files, 124
- include files, 18, 408
- names, 7

**filtering access to contained members, 312****findArea() function, 63, 71–72, 333****float type, 29, 31****floating-point types, 29, 31****flow control with loops**

- breaking out of loops, 81–82
- continuing to next loop, 82–83
- definition of, 79
- do-while loops, 83–84
- for loops, 84–87
- nested loops, 87–88
- switch statements, 89–91
- while loops, 79–81

**Fourteens program, 81****fragility in code, 392****friend classes**

- disadvantages of, 327
- explained, 312–313

**friend functions, pointers to**

- arrays of pointers to functions, 316–318
- arrays of pointers to member functions, 325–327
- declaring, 313–316
- passing to other functions, 319–321
- pointers to member functions, 322–324
- typedef, 321–322

**func() function, 313****funcPtr, 313****function polymorphism, 17, 72****functionality prototypes, 358****FunctionPasser program, 319–321****FunctionPointer program, 314–316****functions. See also specific functions (for example, add() function)**

- accessors, 113, 123
- arguments
  - explained, 21–23
  - passing by reference, 172–174
- auto-typed return values, 73–75
- base class member functions
  - calling, 247–249
  - hiding, 246–247
- compile-time constant expressions, 208–210

**const member functions**

- classes with other classes
  - as member data, 127–132
- declaring, 123
- explained, 123
- function definitions, organizing, 124
- inline implementation, 124–127
- interface versus implementation, 124

**constructors**

- base constructors, passing arguments to, 239–244
- definition of, 117
- inheritance, 236–239
- virtual copy constructors, 260–263

**copy constructors, 204–208, 228****declaring, 61–64****default values, 201–203****defining, 61–64****definition of, 7, 18, 61, 111****destructors**

- explained, 117
- inheritance, 236–239
- virtual destructors, 260

**explained, 20–23****friends. See friend functions, pointers to****function definitions, organizing, 124****function polymorphism, 17****headers, 177****implementing, 114–116****increment methods, 215–216**

- inline anonymous functions, 338–339
- inline functions, 73
- names, 20
- overloading
  - compared to overriding functions, 246
  - explained, 72, 199–201
- overriding
  - compared to function overloading, 246
  - syntax, 244–246
- parameters
  - default parameters, 70–72
  - explained, 21, 67–68
- passing values to, 319–321
- pointers to
  - arrays of pointers to functions, 316–318
  - arrays of pointers to member functions, 325–327
  - declaring, 313–316
  - passing, 319–321
  - pointers to member functions, 322–324
  - typedef, 321–322
- prototypes, 177
- return values
  - auto keyword, 332–334, 339
  - multiple values, 177–179
  - returning by reference, 179–180
- returning values from, 68–70
- signatures, 21

- static member functions
  - advantages of, 327
  - explained, 303–305
- variables
  - explained, 64
  - global variables, 66–67
  - local variables, 64–66
- virtual function members
  - cost of, 263–264
  - declaring, 252–253
  - how they work, 256–257
  - polymorphism, 252–255
  - pure virtual functions, 275–279
  - v-tables (virtual function tables), 256–257
  - when to use, 264

## G

- Garden Gnome Liberation Front, 121**
- gathering requirements, 345**
- GCC compiler, 7**
- GetAge() function, 188**
- getArea() function, 131, 274**
- getArraySize() function, 336**
- getHowMany() function, 305**
- getline() function, 103**
- getPerim() function, 274**
- getSalaryMultiple() function, 332**
- getSpeed() function, 115–116, 123, 125**
- getter functions, 123**

- getUpperLeft() function, 131**
- global name space, 145**
- Global program, 66**
- global variables, 66–67**
- gnomes, lawn, 121**
- “gnoming”, 121**
- gold, panning for, 388**
- goldmaps.com, 388**
- Grader program, 53**
- Granholm, Jackson W., 24**
- grapes, flavor of, 10**
- grasshopper ice cream, 133**
- greater than operator (>), 52**
- greater than or equals operator (>=), 52**
- Griebel, Phillip, 121**

## H

- .h file extension, 124**
- hash (#) symbol, 23**
- Hayes, Woody, 327–328**
- head (linked lists), 287**
- headers
 
  - function headers, 177
  - header files, 124**
- HeadNode class, 290–297**
- heap
 
  - advantages of, 152
  - allocating memory on, 146–147
  - creating objects on, 155
  - deleting objects on, 155–157
  - explained, 145–146**



- member data on heap, 158–160
  - referencing objects on, 192–194
  - restoring memory to, 147–149
  - Heap program, 148**
  - HeapAccessor program, 157–158**
  - HeapCreator program, 156**
  - hiding base class member functions, 246–247
  - high-level design, 345–346
  - history of C++, 5–6, 331
  - horseradish, 211
  - .hp file extension, 124
  - .hpp file extension, 124
  - Hussenot, Francois, 92
  - hyphen (-)
    - decrement operator (--), 47
    - subtraction operator (-), 46
- I**
- IDE (integrated development environment), 9**
  - identifier names, 405–406**
  - if-else**
    - expression values, 58
    - logical operators
      - explained, 56–57
      - logical AND, 56–57
      - logical NOT, 57
      - logical OR, 57
      - relational precedence, 57
  - if-else statements**
    - compound if statements, 54–56
    - else clause, 53–54
    - explained, 52–53
    - when to use, 92
  - implementing. See also initializing**
    - member functions
      - explained, 114–116
      - inline implementation, 124–127
      - interface versus implementation, 124
      - pure virtual functions, 276–279
  - #include directive, 104, 309**
  - include files, 18, 408**
  - increment methods, writing, 215–216**
  - increment operator (++), 47**
  - indirection operator (\*), 140–141**
  - inequality operator (!=), 52**
  - inferring data type, 332–334**
  - infinite loops, 80, 295**
  - inheritance. See also polymorphism**
    - base classes
      - base class initialization, 239–244
      - calling base class member functions, 247–249
      - hiding base class member functions, 246–247
    - constructors, 236–239
    - derivation, 232
    - design, 354–355
    - destructors, 236–239
    - explained, 16, 231
    - overriding functions
      - compared to function overloading, 246
      - syntax, 244–246
    - protected classes, 234–236
    - single inheritance, problems with, 267–271
    - syntax, 233–234
    - when to use, 388
  - initializing**
    - arrays, 98
    - base classes, 239–244
    - multidimensional arrays, 100–101
    - objects, 203–204
    - variables, 33–34
  - inline anonymous functions, 338–339**
  - inline functions, 73**
  - inline implementation of const member functions, 124–127**
  - inline keyword, 73, 124**
  - insert() function, 295–296, 381, 387**
  - instance variables, 111**
  - instances of templates, 374**
  - int() operator, 226–227**
  - integers**
    - binary, 334–335
    - explained, 28, 41
    - signed versus unsigned, 30
    - table of, 31
  - integrated development environment (IDE), 9**
  - interface design prototypes, 357**

**interfaces**

versus implementation, 124  
 PostMaster case study,  
 356–357

**InternalNode class, 290–297**

**International Code of Zoological  
 Nomenclature, 59**

**International Union of Pure and  
 Applied Chemistry (IUPAC), 370**

**interpreters, 13–14**

**isLeapYear() function, 69–70**

**isothiocyanate, 211**

**Istanbul, Turkey, 249**

**iterations, 79**

**IUPAC (International Union of Pure  
 and Applied Chemistry), 370**

**J-K**

***Jagged Little Pill, 228***

**keywords. See also statements**

auto, 38, 62, 332–334, 339  
 catch, 393–397  
 const, 123, 132, 208,  
 247, 408  
 constexpr, 209, 335–338  
 delete, 147–149  
 enum, 37  
 inline, 73, 124  
 new, 146–147  
 private, 113  
 protected, 234  
 public, 112–113  
 template, 374  
 try, 393–397  
 virtual, 252

**kludge, 24**

**Klugman, Jack, 41**

**L**

**lambda expressions, 338–339**

**large groups, programming  
 in, 360**

**late binding, 255**

**lawn gnomes, 121**

**Leak program, 193**

**leaks (memory), 149,  
 192–194, 195**

**LeapYear program, 68–69**

**less than operator (<), 52**

**less than or equals (<=)  
 operator, 52**

**libraries, 7, 356**

**linked lists**

advantages of, 299  
 allocating, 294  
 component parts, 289–297  
 cost of, 298  
 definition of, 287  
 delegation of  
 responsibility, 289  
 doubly linked lists, 287–288  
 explained, 287–288  
 head, 287  
 infinite loops, 295  
 LinkedList program, 290–294  
 nodes, 287  
 as objects, 298–299  
 singly linked lists, 287–288  
 tail, 287

trees, 287–288

weaknesses of, 374

**LinkedList program, 290–294**

**linker, 5–6**

**linking source code, 5–7**

**listings. See programs**

**lists**

linked lists

advantages of, 299  
 allocating, 294  
 component parts,  
 289–297  
 cost of, 298  
 definition of, 287  
 delegation of  
 responsibility, 289  
 doubly linked lists,  
 287–288  
 explained, 287–288  
 head, 287  
 infinite loops, 295  
 LinkedList program,  
 290–294  
 nodes, 287  
 as objects, 298–299  
 singly linked lists, 287–288  
 tail, 287  
 trees, 287–288  
 weaknesses of, 374  
 parameterized List objects,  
 374–381  
 template lists, 381–387

**literal constants**

binary literals, 335  
 numeric literals, 334–335  
 overview, 36

**local variables, 64–66, 75**

**Log() function, 350**

**logic errors, 391**

**logical operators**

explained, 56–57

logical AND, 56–57

logical NOT, 57

logical OR, 57

**long integers, 28, 41**

**long lines of code, 404**

**long long int type, 30, 31**

**long type, 31**

**for loops**

explained, 84–87, 92

foreach, 105–106

**loops**

breaking out of, 81–82

continuing to next loop, 82–83

definition of, 79

do-while, 83–84

event loops, 348–350

for

explained, 84–87

foreach, 105–106

infinite loops, 80, 295

for loops, 92

nested loops, 87–88

switch statements, 89–91

while loops, 79–81, 92

**low-level design, 345–346**

**l-values, 45**

## M

**machine code, 6**

**machine languages, 13**

**main() function, 18–19**

**Mammal1 program, 233–234**

**Mammal2 program, 235–236**

**Mammal3 program, 237–238**

**Mammal4 program, 239–242**

**Mammal5 program, 244–245**

**Mammal6 program, 246–247**

**Mammal7 program, 248**

**Mammal8 program, 252–253**

**Mammal9 program, 253–255**

**Mammal10 program, 257–259**

**Mammal11 program, 260–263**

**Mammal12 program, 267–268**

**Mammal13 program, 269–271**

**Marks, Johnny, 265**

**mathematical operators, 46**

**May, Robert, 264–265**

**member data on heap, 158–160**

**member functions. See functions**

**member variables**

accessing members of contained classes, 311

definition of, 110

static member variables, 301–303

**MemberPointer program, 322–324**

**member-wise (shallow) copies, 204**

**memory**

allocating on heap, 146–147

arrays, 101–102

code space, 145

function pointers

arrays of pointers, 316–318

arrays of pointers to member functions, 325–327

declaring, 313–316

passing, 319–321

pointers to member functions, 322–324

typedef, 321–322

global name space, 145

heap

advantages of, 152

allocating memory on, 146–147

creating objects on, 155

deleting objects on, 155–157

explained, 145–146

member data on heap, 158–160

restoring memory to, 147–149

memory leaks, 149,

192–194, 195

pointers

accessing data members with, 157–158

advantages of, 144–145

const pointers, 162–164, 186–188

dangling pointers, 161–162

declaring, 138

distinction between pointers and addresses, 141

examining addresses stored in, 143–144

explained, 135–138

importance of, 151

indirection operator (\*), 140–141

null pointer constant, 150–151

- null pointers, 138
  - references as alternative to, 189–190
  - storing addresses in, 138–140
  - stray pointers, 161–162
  - this pointer, 160–161
  - when to use, 191
  - wild pointers, 138
- restoring to heap, 147–149
- stack, 145–146
- storing variables in, 28–30
- message format (PostMaster case study), 352–353**
- messages, compiler warnings, 9**
- mistakes. See errors**
- Modern C++, 339**
- modulus operator (%), 46**
- Morissette, Alanis, 228**
- Motto program**
  - comments, 19–20
  - function arguments, 21–23
  - functions, 20–21
  - include files, 18
  - overview, 7–8
  - preprocessor directives, 17–18
  - source code, 18–19
- MPFunction program, 325–326**
- multidimensional arrays**
  - explained, 99–100
  - initializing, 100–101
- multiline comments, 20, 23**
- multiple exception catches, 398**
- multiple values, returning, 177–179**

- multiplication operator (\*), 46**
- MultiTable program, 85–86**

## N

- \n character, 137**

### names

- function names, 20, 62
- identifier names, 405–406
- source code files, 7, 124
- spelling and capitalization, 406
- variable names, 32–33

- National Debt Clock, 335**

- negative numbers, 59**

- nested loops, 87–88**

- new features (C++14)**

- auto keyword in function return types, 332–334
- constexpr keyword, 335–338
- lambda expressions, 338–339
- numeric literals, 334–335
- overview, 331

- new keyword, 146–147**

- NewGrader program, 54–55**

- NewRectangle program, 35**

- NewTricycle program, 118–120**

- nodes (linked lists), 287**

- not equal operator (!=), 52**

- NOT operator (logical), 57**

- Notepad++, 6**

- null character, 102**

- null pointers, 138, 172**

- null references, 172**

- nullptr, 138, 172**

- Number class, 213**

- numbers. See also mathematical operators**

- negative numbers, 59

- Number class, 213

- numeric literals, 334–335

- thirteen, superstitions related to, 107

- numeric literals, 334–335**

## O

- object code, 14**

- object-oriented programming.**

- See OOP (object-oriented programming)

- The Object-Oriented Thought Process, Fourth Edition (Weisfeld), 347***

- ObjectRef program, 184–185**

- objects. See also classes**

- const objects, 165

- creating

- constructors, 116–117

- on heap, 155

- defining, 111–112

- deleting

- destructors, 116–117

- on heap, 155–157

- initializing, 203–204

- linked lists as, 298–299

- references. See also pointers

- address-of operator (&), 169–171

- advantages of, 181

- as alternative to pointers, 189–190
  - creating, 167–168
  - definition of, 167
  - null references, 172
  - objects not in scope, 191–192
  - objects on heap, 192–194
  - passing by reference, 172–174, 183–186
  - returning values by, 179–180
  - swap() function, 175–177
  - what can be referenced, 171–172
  - when to use, 191
  - size of, 121
  - template objects, passing, 381–387
  - Tricycle, 112
- Ohio College of Clowning Arts, 196**
- OOP (object-oriented programming). See also objects; polymorphism**
- advantages of, 369–370
  - classes
    - accessing members of, 112–113
    - class polymorphism, 17
    - constructors, 112–117, 118–120
    - constructors provided by compiler, 118–120
    - contained classes, 305–311
    - declaring, 111
    - default constructors, 112–117
    - definition of, 7, 16
    - explained, 110–111
    - inheritance, 16
    - member functions, 114–116
    - private classes, 112–113, 234
    - protected classes, 234–236
    - public class members, 112–113
  - design. *See also* PostMaster case study
    - analysis and requirements, 345
    - classes, 346–348
    - conceptualization, 344
    - development cycle, 343
    - event loops, 348–350
    - high-level design, 345–346
    - low-level design, 345–346
    - simulation, 344
    - third-party libraries, 356
  - encapsulation
    - definition of, 110
    - explained, 16
  - explained, 15–16
  - inheritance
    - base class initialization, 239–244
    - calling base class member functions, 247–249
    - constructors, 236–239
    - derivation, 232
    - destructors, 236–239
    - explained, 16, 231
    - hiding base class member functions, 246–247
    - overriding functions, 244–246
    - protected classes, 234–236
    - syntax, 233–234
  - objects. *See also* classes
    - creating, 116–117
    - defining, 111–112
    - deleting, 116–117
- operands, 45**
- AND operator (logical), 56–57**
- OR operator (logical), 57**
- operator=, 221–223**
- operators**
- addition operator (+), 219–220
  - address-of operator (&), 169–171
  - assignment operator (=)
    - compared to copy constructor, 228
    - explained, 33, 45–46
    - overloading, 221–223
  - combining, 46–47
  - conversion operators, 224–226
  - decrement operator (--), 47
  - definition of, 45
  - dot operator (.), 112, 157
  - equality operator (==), 224
  - increment operator (++), 47
  - indirection operator (\*), 140–141
  - int() operator, 226–227
  - logical operators
    - logical AND, 56–57
    - logical NOT, 57

- logical OR, 57
- overview, 56–57
- mathematical operators, 46
- output redirection operator (<<), 19
- overloading
  - addition operator (+), 219–220
  - advantages of, 228
  - conversion operators, 224–226
  - equality operator (==), 223
  - explained, 213–214
  - with increment method, 215–216
  - int() operator, 226–227
  - limitations, 220–221
  - operator=, 221–223
  - postfix operator (++), 217–218
  - prefix operator (++), 215–216
- postfix operator (++)
  - explained, 47–49, 228
  - overloading, 217–218
- precedence, 49–51, 57, 59
- prefix operator (++), 47–49, 215–216
- reference operator (&), 169
- relational operators, 51–52
- scope resolution
  - operator (::), 114
- self-assignment operators, 46–47
- order of precedence, 49–51**
- organizing**
  - class definitions, 124
  - function definitions, 124
- output redirection**
- operator (<<), 19**
- Overbey, Charles, 388**
- overloading**
  - constructors, 203
  - destructors, 203
  - functions
    - compared to overriding functions, 246
    - overview, 72
  - member functions, 199–201
  - operators
    - addition operator (+), 219–220
    - advantages of, 228
    - conversion operators, 224–226
    - equality operator (==), 223
    - explained, 213–214
    - with increment method, 215–216
    - int() operator, 226–227
    - limitations, 220–221
    - operator=, 221–223
    - postfix operator (++), 217–218
    - prefix operator (++), 215–216
- overriding functions**
  - compared to function overloading, 246
  - syntax, 244–246
- P**
- pAddress class, 368–369**
- panning for gold, 388**
- parameterized List object, 374–381**
- parameters**
  - compared to arguments, 23
  - default parameters, 70–72
  - explained, 21, 67–68
- ParamList program, 375–380**
- passing values**
  - to base constructors, 239–244
  - const pointers, 186–188
  - contained classes, 312
  - function pointers to other functions, 319–321
  - by reference, 172–174, 183–186
  - template objects, 381–387
  - by value, 186
- pedal() function, 208**
- pencil grades, 76**
- period (.), 103, 112, 157**
- pi, 209–210**
- pID type, 357**
- pObject class, 357**
- Point class, 127–128**
- Pointer program, 142**
- PointerCheck program, 143–144**
- pointers. See also heap; references**
  - accessing data members with, 157–158
  - advantages of, 144–145, 181, 195
  - const pointers, 162–164, 186–188
  - dangling pointers, 161–162
  - declaring, 138

- distinction between pointers and addresses, 141
  - examining addresses stored in, 143–144
  - explained, 135–138
  - to functions
    - arrays of pointers to functions, 316–318
    - arrays of pointers to member functions, 325–327
    - declaring, 313–316
    - passing, 319–321
    - pointers to member functions, 322–324
    - typedef, 321–322
  - importance of, 151
  - indirection operator (\*), 140–141
  - manipulating data with, 141–143
  - memory leaks, 195
  - null pointers, 138, 150–151, 172
  - references as alternative to, 189–190
  - storing addresses in, 138–140
  - stray pointers, 161–162
  - swap() function, 174–175
  - this pointer, 160–161
  - v-pointers, 256–257
  - when to use, 191
  - wild pointers, 138
- PointerSwap program, 174–175**
- PolyException program, 399–402**
- polymorphism. See also inheritance**
- abstract data types
    - advantages of, 284
    - complex hierarchies of abstraction, 279–283
    - explained, 271–275
    - when to use, 283–284
  - exceptions, catching, 398–403
  - explained, 16–17, 251
  - explicit casts, 257
  - single inheritance, problems with, 267–271
  - virtual copy constructors, 260–263
  - virtual destructors, 260
  - virtual function members
    - cost of, 263–264
    - declaring, 252–253
    - how they work, 256–257
    - implementing, 252–255
    - pure virtual functions, 275–279
    - slicing, 257–260
    - v-tables (virtual function tables), 256–257
    - when to use, 264
- polymorphism (function), 72**
- postfix operator (++)**
- explained, 47–49, 228
  - overloading, 217–218
- PostMaster case study**
- 80/20 rule, 358
  - 80/80 rule, 358
  - API (application programming interface), 359–360
  - driver programs, 362–369
  - initial class design, 353–354
  - interfaces, 356–357
  - message format, 352–353
  - ongoing design considerations, 361–362
  - overview, 350–351
  - PostMasterMessage class, 358–359
  - programming in large groups, 360
  - prototype, 357–358
  - rooted versus non-rooted hierarchies, 354–355
  - subprojects, 351–352
- PostMasterMessage class, 358–359**
- PostMasterMessage program, 359–360**
- potassium octanoate, 370**
- pound sign (#), 23**
- precedence (operator), 49–51, 57, 59**
- prefix operator (++)**
- explained, 47–49
  - overloading, 215–216
- preprocessor directives**
- #define, 36
  - #include, 104, 309
  - overview, 17–18
- printError() function, 402**
- private classes, 112–113, 234**
- private keyword, 113**
- procedural programming, 14–15**
- Production program, 106**
- professional-quality code, writing**
- access, 407
  - braces ({}), 403–404
  - class definitions, 407

- comments, 406–407
- const, 408
- identifier names, 405–406
- include files, 408
- long lines, 404
- overview, 403
- program text, 405
- spelling and capitalization of names, 406
- switch statements, 404–405
- program text, writing, 405**
- programming in large groups, 360**
- programming styles, 14–15**
- programs**
  - Addresser, 136–137
  - Animal, 280–283
  - Area, 63
  - AreaCube, 71
  - ArrayFunction, 316–318
  - ArrayMaxer, 337
  - Assignment, 170–171
  - AutoArea, 333
  - AutoCube, 74–75
  - BadTeacher, 90–91
  - Box, 100–101
  - BridgeKeeper, 103–104
  - bulletproof programs, 392
  - Calculator, 21–23
  - Circle, 209
  - Combat, 39
  - Compass, 37–38
  - ConstPasser, 186–188
  - ConstPointer, 163–164
  - Counter, 214
  - Counter2, 215–216
  - Counter3, 217–218
  - Counter4, 219–220
  - Counter5, 224
  - Counter6, 225
  - Counter7, 226–227
  - creating, 7–8
  - DataMember, 158–159
  - DeepCopy, 205–207
  - Driver, 362–368
  - Employee, 309–311
  - Exception, 394–397
  - Expression, 45
  - Fifteens, 82–83
  - Fourteens, 81
  - FunctionPasser, 319–321
  - FunctionPointer, 314–316
  - Global, 66
  - Grader, 53
  - Heap, 148
  - HeapAccessor, 157–158
  - HeapCreator, 156
  - Leak, 193
  - LinkedList, 290–294
  - Mammal1, 233–234
  - Mammal2, 235–236
  - Mammal3, 237–238
  - Mammal4, 239–242
  - Mammal5, 244–245
  - Mammal6, 246–247
  - Mammal7, 248
  - Mammal8, 252–253
  - Mammal9, 253–255
  - Mammal10, 257–259
  - Mammal11, 260–263
  - Mammal12, 267–268
  - Mammal13, 269–271
  - MemberPointer, 322–324
  - Motto
    - comments, 19–20
    - function arguments, 21–23
    - functions, 20–21
    - include files, 18
    - overview, 7–8
    - preprocessor directives, 17–18
    - source code, 18–19
  - MPFunction, 325–326
  - MultTable, 85–86
  - NewGrader, 54–55
  - NewRectangle, 35
  - NewTricycle, 118–120
  - ObjectRef, 184–185
  - ParamList, 375–380
  - Pointer, 142
  - PointerCheck, 143–144
  - PointerSwap, 174–175
  - PolyException, 399–402
  - PostMasterMessage, 359–360
  - Production, 106
  - Rectangle, 33–34, 128–131, 199–200
  - Rectangle2, 201–202
  - Reference, 168
  - Reference2, 169
  - ReferenceSwap, 176
  - RefPasser, 189–190
  - ReturnPointer, 178
  - ReturnRef, 191–192
  - ReturnReference, 179–180



Shape, 272–274  
 Shape2, 276–279  
 SimpleEvent, 348–350  
 Sizer, 29–30  
 StaticFunction, 304–305  
 StaticRobot, 302–303  
 StringCopier, 105  
 structure of  
   comments, 19–20  
   function arguments, 21–23  
   functions, 20–21  
   overview, 17  
   preprocessor directives, 17–18  
   source code, 18–19  
 Swapper, 150–151  
 Temperature, 64–65  
 TemplateList, 381–387  
 Thirteens, 80  
 This, 160–161  
 Tricycle, 114–116  
 ValuePasser, 173  
 Years, 47–49  
**protected classes, 234–236**  
**protected keyword, 234**  
**prototypes**  
   function prototypes, 177  
   PostMaster case study, 357–358  
**public class members, 112–113, 121**  
**public keyword, 112–113**  
**pure virtual functions**  
   declaring, 275–276  
   implementing, 276–279

## Q-R

### quotation marks

“ (double quotes), 137

‘ (single quotes)

  digit separator, 335

  displaying, 137

readability of code, 405

reading arrays with foreach loops, 105–106

Rectangle class, 127–128

Rectangle program, 33–34, 128–131, 199–200

Rectangle2 program, 201–202

red-eye effect (photographs), 165

reference operator (&), 169

Reference program, 168

Reference2 program, 169

references. *See also* pointers

  address-of operator (&), 169–171

  advantages of, 181

  as alternative to pointers, 189–190

  catching exceptions by, 398–403

  creating, 167–168

  definition of, 167

  null references, 172

  objects not in scope, 191–192

  objects on heap, 192–194

  passing by reference

    contained classes, 312

    explained, 183–186

    function arguments, 172–174

  returning values by, 179–180

  swap() function, 175–177

  what can be referenced, 171–172

  when to use, 191

ReferenceSwap program, 176

RefPasser program, 189–190

registering your book, 3

relational operators, 51–52

relational precedence, 57

requirement gathering, 345

responsibility, delegation of, 289

restoring memory to heap, 147–149

return statement, 62

return values

  auto keyword, 332–334, 339

  auto-typed return values, 73–75

  multiple values, 177–179

  obtaining, 68–70

  returning by reference, 179–180

  returning by value, 195

ReturnPointer program, 178

ReturnRef program, 191–192

ReturnReference program, 179–180

reuse through inheritance, 16

rooted versus non-rooted hierarchies, 354–355

“Rudolph the Red-Nosed Reindeer”, 264–265

runtime binding, 255

r-values, 45

**S**

- Sams Teach Yourself UML in 24 Hours, Third Edition* (Schmuller), 345
- Schlichter, Art, 328
- Schuster, Lincoln, 340
- scope
  - explained, 37
  - objects not in scope, 191–192
- scope resolution operator (::), 114
- self-assignment operators, 46–47
- semicolon (;)
  - common mistakes, 86
  - in compound statements, 44
  - with variables, 32
- SetAge() function, 188
- setFirstName() function, 311
- setLastName() function, 311
- setSalary() function, 311
- setSpeed() function, 115–116, 123
- shallow copies, 204
- Shape program, 272–274
- Shape2 program, 276–279
- short integers, 28, 31, 41
- signatures (function), 21
- signed variables, 30
- silbadors, 284–285
- Silbo, 284–285
- Simon, Dick, 340
- SimpleEvent program, 348–350
- simulation, 344
- single inheritance, problems with, 267–271
- single quotes (')
  - digit separator, 335
  - displaying, 137
- singly linked lists, 287–288
- size of objects, 121
- sizeof() function, 29–30, 337–338
- Sizer program, 29–30
- slicing virtual function members, 257–260
- solution space, 345
- source code. *See also* programs
  - compiling, 5–7
  - definition of, 6, 13
  - filenames, 7
  - header files, 124
  - linking, 5–7
- space, 345
- spelling of names, 406
- stack, 145–146
- statements. *See also* loops
  - break, 81–82
  - compound statements, 44
  - continue, 82–83
  - definition of, 43
  - expressions, 44–45
  - if-else
    - compound if statements, 54–56
    - else clause, 53–54
    - explained, 52–53
    - expression values, 58
    - logical operators, 56–57
    - relational precedence, 57
    - when to use, 92
  - return, 62
  - switch, 89–91, 92, 404–405
  - whitespace, 43–44, 59
- static member functions, 303–305, 327
- static member variables, 301–303, 327
- StaticFunction program, 304–305
- StaticRobot program, 302–303
- std=c++14 command-line option, 333
- store() function, 72
- storing
  - addresses in pointers, 138–140
  - variables in memory, 28–30
- stray pointers, 161–162
- strcpy() function, 105
- strcpy\_s() function, 105
- String class, 306–309
- StringCopier program, 105
- strings
  - copying, 104–105
  - definition of, 102
- strncpy() function, 105
- Stroustrup, Bjarne, 5–6, 47, 331, 339
- structured programming, 14–15
- Strupper, Everett, 409
- styles of programming, 14–15
- subtraction operator (-), 46
- swap() function
  - implementing with pointers, 174–175
  - implementing with references, 175–177
  - parameters, 67–68

Swapper, 150–151  
 switch statements, 89–91, 92, 404–405  
 symbolic constants, 36, 41  
 syntactic errors, 391

## T

\t character, 137  
 tail (linked lists), 287  
 TailNode class, 290–297  
 Temperature program, 64–65  
 template keyword, 374  
 TemplateList program, 381–387  
 templates  
   advantages of, 387  
   defining, 374–381  
   explained, 373  
   instances of, 374  
   template objects, passing, 381–387  
   when to use, 388  
 text editors  
   compared to word processors, 9  
   overview, 6  
 third-party libraries, 356  
 thirteen, superstitions related to, 107  
 Thirteens program, 80  
 this pointer, 160–161  
 This program, 160–161  
 throwing exceptions  
   definition of, 393  
   try/catch blocks, 393–397

tilde (~), 117, 203

trees, 287–288

Tricycle class, 111, 125–126, 205–207

Tricycle object, 112

Tricycle program, 114–116

triskaidekaphobia, 107

try blocks, 393–397

type definitions, 34–36

type inference, 332–334

typedef, 34–36, 321–322

types

  abstract data types

    advantages of, 284

    complex hierarchies of abstraction, 279–283

    explained, 271–275

    when to use, 283–284

  auto-typed variables, 38–40

  creating, 110

  explained, 109

  integers

    explained, 28

    signed versus

      unsigned, 30

  pID, 357

  table of, 31

  type definitions, 34–36

  type inference, 332–334

  void, 62

## U

UML (Unified Modeling Language), 345

underscore (\_), 32

Unified Modeling Language (UML), 345

uninitialized array elements, 107

unsigned variables, 30

## V

value, passing by

  contained classes, 312

  explained, 186

ValuePasser program, 173

values

  assigning to variables, 33–34

  default values, 210

  passing by reference

    contained classes, 312

    explained, 172–174, 183–186

  passing by value

    contained classes, 312

    explained, 186

  passing with const pointers, 186–188

  return values

    auto-typed return values, 73–75

    explained, 68–70

    multiple values, 177–179  
 obtaining from functions, 68–70

    returning by reference, 179–180

    returning by value, 195

variables. *See also* pointers

  defining, 32–33

  explained, 27

- with functions
    - explained, 64
    - global variables, 66–67
    - local variables, 64–66
  - initializing, 33–34
  - local variables, 75
  - members of contained
    - classes, accessing, 311
  - names, 32–33
  - signed, 30
  - static member variables, 301–303, 327
  - storing in memory, 28–30
  - types
    - auto-typed variables, 38–40
    - table of, 31
    - type definitions, 34–36
  - unsigned, 30
  - values, assigning, 33–34
  - virtual destructors, 260–263**
  - virtual function members**
    - cost of, 263–264
    - declaring, 252–253
    - how they work, 256–257
    - polymorphism, 252–255
    - pure virtual functions
      - declaring, 275–276
      - implementing, 276–279
    - v-tables (virtual function tables), 256–257
    - when to use, 264
  - virtual function tables (v-tables), 256–257**
  - virtual keyword, 252**
  - Visual Studio Community, 7**
  - void type, 62**
  - v-pointers, 256–257**
  - vptrs, 256–257**
  - v-tables (virtual function tables), 256–257**
- ## W-X-Y-Z
- warnings (compiler), 9
  - Warren, David, 92**
  - wasabi, 211**
  - Washington, George, 152**
  - Washington, Paul Emory, 152**
  - waterfall technique, 343**
  - Web Edition of book, 3**
  - websites**
    - Bjarne Stroustrup, 339
    - C++ standardization website, 340
    - first crossword puzzle, 340
    - goldmaps.com, 388
  - WeightGoals program, 96**
  - Weinberg, Gerald, 391**
  - while loops, 79–81, 92**
  - Whistled Languages* (Busnel and Classe), 284–285**
  - white-eye effect (photographs), 165**
  - whitespace, 43–44, 59, 405**
  - wild pointers, 138, 161–162**
  - word processors, 9**
  - writing**
    - access, 407
    - braces ({}), 403–404
    - class definitions, 407
    - comments, 406–407
    - const, 408
    - identifier names, 405–406
    - include files, 408
    - long lines, 404
    - overview, 403
    - program text, 405
    - spelling and capitalization of names, 406
    - switch statements, 404–405
  - writing past end of a rrays, 97–98**
  - Wynne, Arthur, 340**
  - Years program, 47–49**