Beginning C++ Through Game Programming, Third Edition

**Course Technology PTR**

Michael Dawson

*A part of Cengage Learning*



Australia . Brazil . Japan . Korea . Mexico . Singapore . Spain . United Kingdom . United States



**Beginning C++ Through Game Programming, Third Edition**

**Michael Dawson**

**Publisher and General Manager, Course Technology PTR:** Stacy L. Hiquet

**Associate Director of Marketing:**

Sarah Panella

**Manager of Editorial Services:**

Heather Talbot

**Marketing Manager:** Jordan Castellani **Senior Acquisitions Editor:** Emi Smith **Project Editor:** Jenny Davidson **Technical Reviewer:** Maneesh Sethi

**Interior Layout Tech:** MPS Limited, a Macmillan Company

**Cover Designer:** Mike Tanamachi **Indexer:** Kevin Broccoli **Proofreader:** Michael Beady

© 2011 Course Technology, a part of Cengage Learning.

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

For product information and technology assistance, contact us at

**Cengage Learning Customer & Sales Support,** 1-800-354-9706

For permission to use material from this text or product, submit all requests online at [**www.cengage.com/permissions**](http://www.cengage.com/permissions)Further permissions questions can be emailed to [**permissionrequest@cengage.com**](mailto:permissionrequest@cengage.com)

All trademarks are the property of their respective owners. All images oC Cengage Learning unless otherwise noted.

Library of Congress Control Number: 2010928011

ISBN-13: 978-1-4354-5742-3

ISBN-10: 1-4354-5742-0

eISBN-10:1-4354-5743-9

**Course Technology, a part of Cengage Learning**

20 Channel Center Street Boston, MA 02210

USA

Cengage Learning is a leading provider of customized learning solutions with ofﬁce locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local ofﬁce at: **international.cengage.com/region**

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your lifelong learning solutions, visit **courseptr.com**

Visit our corporate website at **cengage.com**

Printed in the United States of America 1 2 3 4 5 6 7 12 11 10

*To my sweet, tough cookie*—*for all of the help, support, understanding (and distractions) you offered.*

*And to Ariella Saraswati Dawson, a girl who*’*s even more impressive than her name. I look forward to rediscovering the world with you, Monkey.*

## Acknowledgments

Every book you’ve ever read perpetuates a big fat lie. And I’m here to out the publishing industry’s dirty little secret—books are not “by” only one person. Yes, you see only one name on book covers (including this one), but it takes a team of dedicated people to pull off the ﬁnal product. Authors could not do it alone; I certainly could not have done it alone. So I want to thank all those who helped make this book a reality.

Thanks to Jenny Davidson for her dual role as Project Editor and Copy Editor. Jenny kept me on schedule and my commas in place.

Thanks to Maneesh Sethi, my Technical Reviewer, who made sure my programs worked the way I said they did.

Thanks to Michael Beady, my Proofreader. His work makes this book look good—literally.

I also want to thank Emi Smith, my Senior Acquisitions Editor, for all of her encouragement.

Finally, I want to thank all of the game programmers who created the games I played while growing up. They inspired me to work in the industry and create games of my own. I hope I can inspire a few readers to do the same.

## About the Author

**Michael Dawson** is a game programming author and instructor who teaches students the art and science of writing their own games. Mike has developed and taught game programming courses for UCLA Extension, The Digital Media Academy, and The Los Angeles Film School. In addition, his books have been required reading in colleges and universities around the country.

Mike got his start in the game industry as a producer and designer, but he also “starred” in an adventure game in which the player controls the main character, named Mike Dawson. In the game, the player directs the digitized images of Dawson, who must stop an extraterrestrial invasion before an implanted alien embryo is born from his head.

In real life, Mike is the author of *Beginning C*þþ *Through Game Programming*, *Python Programming for the Absolute Beginner*, *C*þþ *Projects: Programming with Text-Based Games*, and *Guide to Programming with Python*. He earned his bachelor’s degree in Computer Science from the University of Southern California. Visit his website at [www.programgames.com](http://www.programgames.com/) to learn more or to get support for any of his books.

## Contents

[Introduction xviii](#_bookmark13)

|  |  |  |  |
| --- | --- | --- | --- |
| [**Chapter**](#_bookmark15) | [**1**](#_bookmark15) | [**Types, Variables, and Standard I/O: Lost Fortune . . . . . . . .**](#_bookmark15) | [**1**](#_bookmark15) |
|  |  | [Introducing Cþþ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark17) | [1](#_bookmark17) |
|  |  | [Using Cþþ for Games . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark18) | [2](#_bookmark18) |
|  |  | [Creating an Executable File . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark20) | [2](#_bookmark20) |
|  |  | [Dealing with Errors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark22) | [4](#_bookmark22) |
|  |  | [Understanding the ISO Standard . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark24) | [5](#_bookmark24) |
|  |  | [Writing Your First Cþþ Program . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark26) | [5](#_bookmark26) |
|  |  | [Introducing the Game Over Program . . . . . . . . . . . . . . . . . . . . . .](#_bookmark27) | [5](#_bookmark27) |
|  |  | [Commenting Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark29) | [7](#_bookmark29) |
|  |  | [Using Whitespace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark32) | [7](#_bookmark32) |
|  |  | [Including Other Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark33) | [7](#_bookmark33) |
|  |  | [Defining the main() Function . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark34) | [8](#_bookmark34) |
|  |  | [Displaying Text through the Standard Output . . . . . . . . . . . . . . .](#_bookmark37) | [8](#_bookmark37) |
|  |  | [Terminating Statements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark39) | [9](#_bookmark39) |
|  |  | [Returning a Value from main() . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark40) | [10](#_bookmark40) |
|  |  | [Working with the std Namespace . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark42) | [10](#_bookmark42) |
|  |  | [Introducing the Game Over 2.0 Program . . . . . . . . . . . . . . . . . .](#_bookmark43) | [10](#_bookmark43) |
|  |  | [Employing a using Directive . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark44) | [11](#_bookmark44) |
|  |  | [Introducing the Game Over 3.0 Program . . . . . . . . . . . . . . . . . .](#_bookmark46) | [11](#_bookmark46) |
|  |  | [Employing using Declarations . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark47) | [12](#_bookmark47) |
|  |  | [Understanding When to Employ using . . . . . . . . . . . . . . . . . . . .](#_bookmark49) | [12](#_bookmark49) |

vi

Contents vii

[Using Arithmetic Operators 13](#_bookmark50)

[Introducing the Expensive Calculator Program 13](#_bookmark51)

[Adding, Subtracting, and Multiplying 14](#_bookmark53)

[Understanding Integer and Floating Point Division 14](#_bookmark55)

[Using the Modulus Operator 15](#_bookmark56)

[Understanding Order of Operations 15](#_bookmark58)

[Declaring and Initializing Variables 16](#_bookmark59)

[Introducing the Game Stats Program 16](#_bookmark60)

[Understanding Fundamental Types 18](#_bookmark63)

[Understanding Type Modifiers 18](#_bookmark64)

[Declaring Variables 19](#_bookmark67)

[Naming Variables 20](#_bookmark70)

[Assigning Values to Variables 21](#_bookmark73)

[Initializing Variables 22](#_bookmark75)

[Displaying Variable Values 22](#_bookmark77)

[Getting User Input 23](#_bookmark78)

[Defining New Names for Types 23](#_bookmark80)

[Understanding Which Types to Use 24](#_bookmark81)

[Performing Arithmetic Operations with Variables 24](#_bookmark82)

[Introducing the Game Stats 2.0 Program 24](#_bookmark84)

[Altering the Value of a Variable 26](#_bookmark86)

[Using Combined Assignment Operators 26](#_bookmark87)

[Using Increment and Decrement Operators 27](#_bookmark88)

[Dealing with Integer Wrap Around 28](#_bookmark91)

[Working with Constants 29](#_bookmark92)

[Introducing the Game Stats 3.0 Program 29](#_bookmark94)

[Using Constants 31](#_bookmark96)

[Using Enumerations 31](#_bookmark98)

[Introducing Lost Fortune 32](#_bookmark100)

[Setting Up the Program 32](#_bookmark101)

[Getting Information from the Player 33](#_bookmark103)

[Telling the Story 34](#_bookmark105)

[Summary 35](#_bookmark108)

[Questions and Answers 36](#_bookmark111)

[Discussion Questions 38](#_bookmark113)

[Exercises 38](#_bookmark115)

[**Chapter 2 Truth, Branching, and the Game Loop:**](#_bookmark116)

[**Guess My Number 39**](#_bookmark116)

[Understanding Truth 39](#_bookmark118)

|  |  |  |  |
| --- | --- | --- | --- |
| viii | Contents |  | |
|  |  | [Using the if Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark121) | [40](#_bookmark121) |
|  |  | [Introducing the Score Rater Program . . . . . . . . . . . . . . . . . . . . .](#_bookmark122) | [41](#_bookmark122) |
|  |  | [Testing true and false . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark125) | [42](#_bookmark125) |
|  |  | [Interpreting a Value as true or false . . . . . . . . . . . . . . . . . . . . .](#_bookmark127) | [43](#_bookmark127) |
|  |  | [Using Relational Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark128) | [44](#_bookmark128) |
|  |  | [Nesting if Statements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark130) | [44](#_bookmark130) |
|  |  | [Using the else Clause . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark132) | [45](#_bookmark132) |
|  |  | [Introducing the Score Rater 2.0 Program . . . . . . . . . . . . . . . . . .](#_bookmark133) | [46](#_bookmark133) |
|  |  | [Creating Two Ways to Branch . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark136) | [47](#_bookmark136) |
|  |  | [Using a Sequence of if Statements with else Clauses . . . . . . . . . . . .](#_bookmark137) | [48](#_bookmark137) |
|  |  | [Introducing the Score Rater 3.0 Program . . . . . . . . . . . . . . . . . .](#_bookmark139) | [49](#_bookmark139) |
|  |  | [Creating a Sequence of if Statements with else Clauses . . . . . . .](#_bookmark142) | [50](#_bookmark142) |
|  |  | [Using the switch Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark143) | [51](#_bookmark143) |
|  |  | [Introducing the Menu Chooser Program . . . . . . . . . . . . . . . . . .](#_bookmark145) | [52](#_bookmark145) |
|  |  | [Creating Multiple Ways to Branch . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark148) | [54](#_bookmark148) |
|  |  | [Using while Loops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark149) | [54](#_bookmark149) |
|  |  | [Introducing the Play Again Program . . . . . . . . . . . . . . . . . . . . .](#_bookmark151) | [54](#_bookmark151) |
|  |  | [Looping with a while Loop . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark153) | [55](#_bookmark153) |
|  |  | [Using do Loops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark154) | [56](#_bookmark154) |
|  |  | [Introducing the Play Again 2.0 Program . . . . . . . . . . . . . . . . . .](#_bookmark156) | [56](#_bookmark156) |
|  |  | [Looping with a do Loop . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark158) | [57](#_bookmark158) |
|  |  | [Using break and continue Statements . . . . . . . . . . . . . . . . . . . . . .](#_bookmark160) | [58](#_bookmark160) |
|  |  | [Introducing the Finicky Counter Program . . . . . . . . . . . . . . . . . .](#_bookmark161) | [58](#_bookmark161) |
|  |  | [Creating a while (true) Loop . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark163) | [60](#_bookmark163) |
|  |  | [Using the break Statement to Exit a Loop . . . . . . . . . . . . . . . . .](#_bookmark165) | [60](#_bookmark165) |
|  |  | [Using the continue Statement to Jump Back](#_bookmark166) |  |
|  |  | [to the Top of a Loop . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark166) | [61](#_bookmark166) |
|  |  | [Understanding When to Use break and continue . . . . . . . . . . . .](#_bookmark167) | [61](#_bookmark167) |
|  |  | [Using Logical Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark170) | [61](#_bookmark170) |
|  |  | [Introducing the Designers Network Program . . . . . . . . . . . . . . .](#_bookmark171) | [62](#_bookmark171) |
|  |  | [Using the Logical AND Operator . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark173) | [65](#_bookmark173) |
|  |  | [Using the Logical OR Operator . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark175) | [66](#_bookmark175) |
|  |  | [Using the Logical NOT Operator . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark177) | [66](#_bookmark177) |
|  |  | [Understanding Order of Operations . . . . . . . . . . . . . . . . . . . . .](#_bookmark179) | [67](#_bookmark179) |
|  |  | [Generating Random Numbers . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark180) | [68](#_bookmark180) |
|  |  | [Introducing the Die Roller Program . . . . . . . . . . . . . . . . . . . . . .](#_bookmark182) | [68](#_bookmark182) |
|  |  | [Calling the rand() Function . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark184) | [69](#_bookmark184) |
|  |  | [Seeding the Random Number Generator . . . . . . . . . . . . . . . . . .](#_bookmark186) | [70](#_bookmark186) |
|  |  | [Calculating a Number within a Range . . . . . . . . . . . . . . . . . . . .](#_bookmark188) | [71](#_bookmark188) |

[Contents ix](#_TOC_250000)

[Understanding the Game Loop 72](#_bookmark189)

[Introducing Guess My Number 73](#_bookmark192)

[Applying the Game Loop 74](#_bookmark193)

[Setting Up the Game 74](#_bookmark195)

[Creating the Game Loop 76](#_bookmark197)

[Wrapping Up the Game 76](#_bookmark199)

[Summary 76](#_bookmark200)

[Questions and Answers 78](#_bookmark202)

[Discussion Questions 80](#_bookmark205)

[Exercises 80](#_bookmark206)

[Chapter 3 For Loops, Strings, and Arrays: Word Jumble 81](#_bookmark207)

[Using for Loops 81](#_bookmark209)

[Introducing the Counter Program 82](#_bookmark210)

[Counting with for Loops 84](#_bookmark213)

[Using Empty Statements in for Loops 85](#_bookmark215)

[Nesting for Loops 86](#_bookmark216)

[Understanding Objects 87](#_bookmark218)

[Using String Objects 89](#_bookmark221)

[Introducing the String Tester Program 89](#_bookmark223)

[Creating string Objects 91](#_bookmark225)

[Concatenating string Objects 92](#_bookmark226)

[Using the size() Member Function 92](#_bookmark228)

[Indexing a string Object 93](#_bookmark229)

[Iterating through string Objects 93](#_bookmark231)

[Using the find() Member Function 94](#_bookmark232)

[Using the erase() Member Function 95](#_bookmark234)

[Using the empty() Member Function 96](#_bookmark236)

[Using Arrays 96](#_bookmark237)

[Introducing the Hero’s Inventory Program 96](#_bookmark239)

[Creating Arrays 98](#_bookmark242)

[Indexing Arrays 99](#_bookmark244)

[Accessing Member Functions of an Array Element 100](#_bookmark246)

[Being Aware of Array Bounds 100](#_bookmark247)

[Understanding C-Style Strings 101](#_bookmark249)

[Using Multidimensional Arrays 103](#_bookmark251)

[Introducing the Tic-Tac-Toe Board Program 103](#_bookmark252)

[Creating Multidimensional Arrays 105](#_bookmark254)

[Indexing Multidimensional Arrays 105](#_bookmark256)

x Contents

[Introducing Word Jumble 106](#_bookmark257)

[Setting Up the Program 107](#_bookmark259)

[Picking a Word to Jumble 107](#_bookmark261)

[Jumbling the Word 108](#_bookmark263)

[Welcoming the Player 109](#_bookmark264)

[Entering the Game Loop 109](#_bookmark266)

[Saying Goodbye 110](#_bookmark267)

[Summary 110](#_bookmark270)

[Questions and Answers 111](#_bookmark272)

[Discussion Questions 113](#_bookmark275)

[Exercises 114](#_bookmark276)

[Chapter 4 The Standard Template Library: Hangman 115](#_bookmark277)

[Introducing the Standard Template Library 115](#_bookmark279)

[Using Vectors 116](#_bookmark280)

[Introducing the Hero’s Inventory 2.0 Program 117](#_bookmark283)

[Preparing to Use Vectors 119](#_bookmark285)

[Declaring a Vector 119](#_bookmark287)

[Using the push\_back() Member Function 120](#_bookmark288)

[Using the size() Member Function 120](#_bookmark290)

[Indexing Vectors 121](#_bookmark291)

[Calling Member Functions of an Element 121](#_bookmark293)

[Using the pop\_back() Member Function 122](#_bookmark294)

[Using the clear() Member Function 122](#_bookmark295)

[Using the empty() Member Function 122](#_bookmark298)

[Using Iterators 123](#_bookmark299)

[Introducing the Hero’s Inventory 3.0 Program 123](#_bookmark300)

[Declaring Iterators 125](#_bookmark303)

[Looping through a Vector 126](#_bookmark305)

[Changing the Value of a Vector Element 128](#_bookmark308)

[Accessing Member Functions of a Vector Element 129](#_bookmark309)

[Using the insert() Vector Member Function 130](#_bookmark311)

[Using the erase() Vector Member Function 130](#_bookmark313)

[Using Algorithms 131](#_bookmark314)

[Introducing the High Scores Program 131](#_bookmark316)

[Preparing to Use Algorithms 133](#_bookmark318)

[Using the find() Algorithm 134](#_bookmark319)

[Using the random\_shuffle() Algorithm 134](#_bookmark321)

[Using the sort() Algorithm 135](#_bookmark322)

Contents xi

[Understanding Vector Performance 136](#_bookmark324)

[Examining Vector Growth 136](#_bookmark325)

[Examining Element Insertion and Deletion 138](#_bookmark329)

[Examining Other STL Containers 138](#_bookmark332)

[Planning Your Programs 139](#_bookmark334)

[Using Pseudocode 139](#_bookmark335)

[Using Stepwise Refinement 140](#_bookmark338)

[Introducing Hangman 141](#_bookmark339)

[Planning the Game 141](#_bookmark342)

[Setting Up the Program 142](#_bookmark344)

[Initializing Variables and Constants 143](#_bookmark345)

[Entering the Main Loop 143](#_bookmark347)

[Getting the Player’s Guess 144](#_bookmark348)

[Ending the Game 145](#_bookmark350)

[Summary 145](#_bookmark353)

[Questions and Answers 146](#_bookmark356)

[Discussion Questions 148](#_bookmark359)

[Exercises 148](#_bookmark360)

[Chapter 5 Functions: Mad Lib 151](#_bookmark362)

[Creating Functions 151](#_bookmark364)

[Introducing the Instructions Program 152](#_bookmark365)

[Declaring Functions 153](#_bookmark367)

[Defining Functions 154](#_bookmark369)

[Calling Functions 154](#_bookmark372)

[Understanding Abstraction 155](#_bookmark373)

[Using Parameters and Return Values 155](#_bookmark375)

[Introducing the Yes or No Program 155](#_bookmark376)

[Returning a Value 157](#_bookmark378)

[Accepting Values into Parameters 158](#_bookmark380)

[Understanding Encapsulation 160](#_bookmark382)

[Understanding Software Reuse 161](#_bookmark384)

[Working with Scopes 161](#_bookmark386)

[Introducing the Scoping Program 161](#_bookmark387)

[Working with Separate Scopes 163](#_bookmark389)

[Working with Nested Scopes 165](#_bookmark391)

[Using Global Variables 166](#_bookmark394)

[Introducing the Global Reach Program 166](#_bookmark397)

[Declaring Global Variables 168](#_bookmark398)

[Accessing Global Variables 168](#_bookmark400)

xii Contents

[Hiding Global Variables 169](#_bookmark401)

[Altering Global Variables 169](#_bookmark403)

[Minimizing the Use of Global Variables 170](#_bookmark404)

[Using Global Constants 170](#_bookmark405)

[Using Default Arguments 171](#_bookmark407)

[Introducing the Give Me a Number Program 171](#_bookmark408)

[Specifying Default Arguments 173](#_bookmark411)

[Assigning Default Arguments to Parameters 173](#_bookmark413)

[Overriding Default Arguments 174](#_bookmark414)

[Overloading Functions 174](#_bookmark417)

[Introducing the Triple Program 174](#_bookmark418)

[Creating Overloaded Functions 176](#_bookmark420)

[Calling Overloaded Functions 177](#_bookmark422)

[Inlining Functions 177](#_bookmark423)

[Introducing the Taking Damage Program 177](#_bookmark426)

[Specifying Functions for Inlining 179](#_bookmark427)

[Calling Inlined Functions 179](#_bookmark429)

[Introducing the Mad Lib Game 180](#_bookmark430)

[Setting Up the Program 181](#_bookmark432)

[The main() Function 181](#_bookmark434)

[The askText() Function 182](#_bookmark435)

[The askNumber() Function 182](#_bookmark437)

[The tellStory() Function 183](#_bookmark438)

[Summary 183](#_bookmark440)

[Questions and Answers 184](#_bookmark443)

[Discussion Questions 186](#_bookmark446)

[Exercises 186](#_bookmark448)

[Chapter 6 References: Tic-Tac-Toe 187](#_bookmark449)

[Using References 187](#_bookmark451)

[Introducing the Referencing Program 187](#_bookmark452)

[Creating References 189](#_bookmark453)

[Accessing Referenced Values 190](#_bookmark455)

[Altering Referenced Values 190](#_bookmark457)

[Passing References to Alter Arguments 191](#_bookmark458)

[Introducing the Swap Program 191](#_bookmark460)

[Passing by Value 193](#_bookmark463)

[Passing by Reference 194](#_bookmark464)

[Passing References for Efficiency 195](#_bookmark466)

[Introducing the Inventory Displayer Program 195](#_bookmark467)

Contents xiii

[Understanding the Pitfalls of Reference Passing 196](#_bookmark470)

[Declaring Parameters as Constant References 197](#_bookmark471)

[Passing a Constant Reference 197](#_bookmark473)

[Deciding How to Pass Arguments 198](#_bookmark474)

[Returning References 198](#_bookmark476)

[Introducing the Inventory Referencer Program 199](#_bookmark477)

[Returning a Reference 200](#_bookmark480)

[Displaying the Value of a Returned Reference 201](#_bookmark482)

[Assigning a Returned Reference to a Reference 202](#_bookmark483)

[Assigning a Returned Reference to a Variable 202](#_bookmark484)

[Altering an Object through a Returned Reference 202](#_bookmark486)

[Introducing the Tic-Tac-Toe Game 203](#_bookmark487)

[Planning the Game 203](#_bookmark489)

[Setting Up the Program 205](#_bookmark492)

[The main() Function 207](#_bookmark494)

[The instructions() Function 208](#_bookmark496)

[The askYesNo() Function 208](#_bookmark498)

[The askNumber() Function 209](#_bookmark499)

[The humanPiece() Function 209](#_bookmark501)

[The opponent() Function 210](#_bookmark502)

[The displayBoard() Function 210](#_bookmark504)

[The winner() Function 211](#_bookmark505)

[The isLegal() Function 212](#_bookmark508)

[The humanMove() Function 213](#_bookmark509)

[The computerMove() Function 213](#_bookmark511)

[The announceWinner() Function 217](#_bookmark513)

[Summary 217](#_bookmark516)

[Questions and Answers 218](#_bookmark518)

[Discussion Questions 220](#_bookmark521)

[Exercises 221](#_bookmark522)

[Chapter 7 Pointers: Tic-Tac-Toe 2.0 223](#_bookmark523)

[Understanding Pointer Basics 223](#_bookmark525)

[Introducing the Pointing Program 224](#_bookmark526)

[Declaring Pointers 226](#_bookmark528)

[Initializing Pointers 227](#_bookmark531)

[Assigning Addresses to Pointers 227](#_bookmark534)

[Dereferencing Pointers 228](#_bookmark537)

[Reassigning Pointers 229](#_bookmark540)

[Using Pointers to Objects 230](#_bookmark541)

xiv Contents

[Understanding Pointers and Constants 231](#_bookmark543)

[Using a Constant Pointer 231](#_bookmark544)

[Using a Pointer to a Constant 232](#_bookmark546)

[Using a Constant Pointer to a Constant 233](#_bookmark548)

[Summarizing Constants and Pointers 234](#_bookmark549)

[Passing Pointers 234](#_bookmark551)

[Introducing the Swap Pointer Version Program 234](#_bookmark552)

[Passing by Value 236](#_bookmark554)

[Passing a Constant Pointer 237](#_bookmark555)

[Returning Pointers 238](#_bookmark558)

[Introducing the Inventory Pointer Program 239](#_bookmark559)

[Returning a Pointer 240](#_bookmark562)

[Using a Returned Pointer to Display a Value 241](#_bookmark564)

[Assigning a Returned Pointer to a Pointer 242](#_bookmark565)

[Assigning to a Variable the Value Pointed](#_bookmark567)

[to by a Returned Pointer 242](#_bookmark567)

[Altering an Object through a Returned Pointer 243](#_bookmark569)

[Understanding the Relationship between Pointers and Arrays 244](#_bookmark570)

[Introducing the Array Passer Program 244](#_bookmark572)

[Using an Array Name as a Constant Pointer 246](#_bookmark573)

[Passing and Returning Arrays 247](#_bookmark575)

[Introducing the Tic-Tac-Toe 2.0 Game 248](#_bookmark577)

[Summary 248](#_bookmark580)

[Questions and Answers 250](#_bookmark583)

[Discussion Questions 252](#_bookmark586)

[Exercises 252](#_bookmark587)

[Chapter 8 Classes: Critter Caretaker 255](#_bookmark588)

[Defining New Types 255](#_bookmark590)

[Introducing the Simple Critter Program 256](#_bookmark591)

[Defining a Class 257](#_bookmark594)

[Defining Member Functions 258](#_bookmark597)

[Instantiating Objects 259](#_bookmark598)

[Accessing Data Members 259](#_bookmark601)

[Calling Member Functions 260](#_bookmark602)

[Using Constructors 260](#_bookmark604)

[Introducing the Constructor Critter Program 261](#_bookmark605)

[Declaring and Defining a Constructor 262](#_bookmark608)

[Calling a Constructor Automatically 263](#_bookmark610)

Contents xv

[Setting Member Access Levels 264](#_bookmark611)

[Introducing the Private Critter Program 264](#_bookmark612)

[Specifying Public and Private Access Levels 266](#_bookmark615)

[Defining Accessor Member Functions 267](#_bookmark617)

[Defining Constant Member Functions 268](#_bookmark619)

[Using Static Data Members and Member Functions 269](#_bookmark621)

[Introducing the Static Critter Program 270](#_bookmark622)

[Declaring and Initializing Static Data Members 272](#_bookmark625)

[Accessing Static Data Members 272](#_bookmark627)

[Declaring and Defining Static Member Functions 273](#_bookmark628)

[Calling Static Member Functions 273](#_bookmark631)

[Introducing the Critter Caretaker Game 274](#_bookmark632)

[Planning the Game 275](#_bookmark634)

[Planning the Pseudocode 276](#_bookmark638)

[The Critter Class 277](#_bookmark639)

[The main() Function 280](#_bookmark645)

[Summary 281](#_bookmark647)

[Questions and Answers 283](#_bookmark650)

[Discussion Questions 285](#_bookmark654)

[Exercises 285](#_bookmark655)

|  |  |  |  |
| --- | --- | --- | --- |
| [**Chapter**](#_bookmark656) | [**9**](#_bookmark656) | [**Advanced Classes and Dynamic Memory:**](#_bookmark656) |  |
|  |  | [**Game Lobby . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .**](#_bookmark656) | [**287**](#_bookmark656) |
|  |  | [Using Aggregation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark658) | [287](#_bookmark658) |
|  |  | [Introducing the Critter Farm Program . . . . . . . . . . . . . . . . . . .](#_bookmark659) | [288](#_bookmark659) |
|  |  | [Using Object Data Members . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark662) | [290](#_bookmark662) |
|  |  | [Using Container Data Members . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark664) | [291](#_bookmark664) |
|  |  | [Using Friend Functions and Operator Overloading . . . . . . . . . . . .](#_bookmark666) | [292](#_bookmark666) |
|  |  | [Introducing the Friend Critter Program . . . . . . . . . . . . . . . . . .](#_bookmark667) | [292](#_bookmark667) |
|  |  | [Creating Friend Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark669) | [295](#_bookmark669) |
|  |  | [Overloading Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark672) | [295](#_bookmark672) |
|  |  | [Dynamically Allocating Memory . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark675) | [296](#_bookmark675) |
|  |  | [Introducing the Heap Program . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark676) | [297](#_bookmark676) |
|  |  | [Using the new Operator . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark678) | [299](#_bookmark678) |
|  |  | [Using the delete Operator . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark681) | [300](#_bookmark681) |
|  |  | [Avoiding Memory Leaks . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark683) | [301](#_bookmark683) |
|  |  | [Working with Data Members and the Heap . . . . . . . . . . . . . . . . .](#_bookmark684) | [303](#_bookmark684) |
|  |  | [Introducing the Heap Data Member Program . . . . . . . . . . . . .](#_bookmark686)  [Declaring Data Members that Point to](#_bookmark687)  [Values on the Heap . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .](#_bookmark687) | [303](#_bookmark686)  [307](#_bookmark687) |

xvi Contents

[Declaring and Defining Destructors 308](#_bookmark689)

[Declaring and Defining Copy Constructors 309](#_bookmark692)

[Overloading the Assignment Operator 313](#_bookmark696)

[Introducing the Game Lobby Program 315](#_bookmark700)

[The Player Class 316](#_bookmark702)

[The Lobby Class 318](#_bookmark704)

[The Lobby::AddPlayer() Member Function 320](#_bookmark706)

[The Lobby::RemovePlayer() Member Function 322](#_bookmark708)

[The Lobby::Clear() Member Function 322](#_bookmark710)

[The operator*<<*() Member Function 323](#_bookmark712)

[The main() Function 324](#_bookmark714)

[Summary 325](#_bookmark717)

[Questions and Answers 326](#_bookmark719)

[Discussion Questions 328](#_bookmark720)

[Exercises 328](#_bookmark721)

[**Chapter 10 Inheritance and Polymorphism: Blackjack 331**](#_bookmark722)

[Introducing Inheritance 331](#_bookmark724)

[Introducing the Simple Boss Program 333](#_bookmark726)

[Deriving from a Base Class 335](#_bookmark729)

[Instantiating Objects from a Derived Class 336](#_bookmark731)

[Using Inherited Members 337](#_bookmark732)

[Controlling Access under Inheritance 337](#_bookmark734)

[Introducing the Simple Boss 2.0 Program 338](#_bookmark735)

[Using Access Modifiers with Class Members 339](#_bookmark739)

[Using Access Modifiers when Deriving Classes 340](#_bookmark740)

[Calling and Overriding Base Class Member Functions 340](#_bookmark743)

[Introducing the Overriding Boss Program 341](#_bookmark744)

[Calling Base Class Constructors 343](#_bookmark748)

[Declaring Virtual Base Class Member Functions 344](#_bookmark749)

[Overriding Virtual Base Class Member Functions 344](#_bookmark752)

[Calling Base Class Member Functions 345](#_bookmark754)

[Using Overloaded Assignment Operators and](#_bookmark756)

[Copy Constructors in Derived Classes 346](#_bookmark756)

[Introducing Polymorphism 347](#_bookmark757)

[Introducing the Polymorphic Bad Guy Program 347](#_bookmark759)

[Using Base Class Pointers to Derived Class Objects 350](#_bookmark760)

[Defining Virtual Destructors 351](#_bookmark763)

[Using Abstract Classes 352](#_bookmark765)

[Introducing the Abstract Creature Program 352](#_bookmark766)

Contents xvii

[Declaring Pure Virtual Functions 354](#_bookmark768)

[Deriving a Class from an Abstract Class 355](#_bookmark769)

[Introducing the Blackjack Game 356](#_bookmark771)

[Designing the Classes 356](#_bookmark773)

[Planning the Game Logic 360](#_bookmark777)

[The Card Class 361](#_bookmark779)

[The Hand Class 363](#_bookmark781)

[The GenericPlayer Class 366](#_bookmark784)

[The Player Class 368](#_bookmark786)

[The House Class 369](#_bookmark789)

[The Deck Class 370](#_bookmark791)

[The Game Class 373](#_bookmark794)

[The main() Function 376](#_bookmark796)

[Overloading the operator*<<*() Function 377](#_bookmark798)

[Summary 379](#_bookmark799)

[Questions and Answers 380](#_bookmark803)

[Discussion Questions 382](#_bookmark805)

[Exercises 382](#_bookmark807)

[**Appendix A Creating Your First C++ Program 383**](#_bookmark808)

[**Appendix B Operator Precedence 389**](#_bookmark814)

[**Appendix C Keywords 391**](#_bookmark817)

[**Appendix D ASCII Chart 393**](#_bookmark820)

[**Appendix E Escape Sequences 397**](#_bookmark823)

[**Index 399**](#_bookmark825)

## [Introduction](#_bookmark0)

Cutting-edge computer games rival the best that Hollywood has to offer in visual effects, musical score, and pure adrenaline rush. But games are a form of enter- tainment unlike any other; they can keep players glued to their monitors for hours on end. What sets games apart and makes them so engrossing is interactivity. In a computer game, you don’t simply sit back and watch a hero ﬁghting against all odds, you *become* the hero.

The key to achieving this interactivity is programming. It’s programming that allows an alien creature, an attack squadron, or an entire army to react differently to a player in different situations. Through programming, a game’s story can unfold in new ways. In fact, as the result of programming, a game can respond to a player in ways that the game creators might never have imagined.

Although there are literally thousands of computer programming languages, Cþþ is the game industry standard. If you were to wander the PC game section of your favorite store and grab a title at random, the odds are overwhelming that the game in your hand would be written largely or exclusively in Cþþ. The bottom line is this: If you want to program computer games professionally, you must know Cþþ.

The goal of this book is to introduce you to the Cþþ language from a game programming perspective. Although no single book can make you the master of two deep topics such as Cþþ and game programming, this book will start you on your journey.

xviii

Introduction xix

### Who This Book Is For

This book is for anyone who wants to program games. It’s aimed at the total beginner and assumes no previous programming experience. If you’re comfor- table using your computer, then you can start your game programming odyssey right here. But just because this book is written for the beginner, that doesn’t mean learning Cþþ and game programming will be easy. You’ll have to read, work, and experiment. By the end of this book, you’ll have a solid foundation in the game programming language of the professionals.

### How This Book Is Organized

I start at the very beginning of Cþþ and game programming, assuming no experience in either. As the chapters progress, I cover more advanced topics, building on previous material.

In each chapter, I cover one or several related topics. I move through concepts one step at a time by writing bite-sized, game-related programs to demonstrate each idea. At the end of each chapter, I combine some of the most important concepts in a single game. The last chapter of the book ends with the most ambitious project—one that harnesses all of the major concepts presented throughout the book.

In addition to learning about Cþþ and game programming, you’ll also learn how to organize your work, break down problems into manageable chunks, and reﬁne your code. You’ll be challenged at times, but never overwhelmed. Most of all, you’ll have fun while learning. In the process, you’ll create some cool computer games and gain insight into the craft of game programming.

**Chapter 1: Types, Variables, and Standard I/O: Lost Fortune.** You’ll be introduced to the fundamentals of Cþþ, the standard language of the game industry. You’ll learn to display output in a console window, perform arithmetic computations, use variables, and get player input from the keyboard.

**Chapter 2: Truth, Branching, and the Game Loop: Guess My Number.** You’ll create more interesting games by writing programs that execute, skip, or repeat sections of code based on some condition. You’ll learn how to generate random numbers to add some unpredictability to your games. And you’ll learn about the Game Loop—a fundamental way to organize your games to keep the action going.

xx Introduction

**Chapter 3: For Loops, Strings, and Arrays: Word Jumble.** You’ll learn about sequences and work with strings—sequences of characters that are perfect for word games. You also learn about software objects—entities that can be used to represent objects in your games, such as alien spacecrafts, healing potions, or even the player himself.

**Chapter 4: The Standard Template Library: Hangman.** You’ll be introduced to a powerful library—a toolbox that game programmers (and even non-game programmers) rely on to hold collections of things, such as items in a player’s inventory. You’ll also learn about techniques that can help you plan larger game programs.

**Chapter 5: Functions: Mad Lib.** You’ll learn to break up your game programs into smaller, more manageable chunks of code. You’ll accomplish this by dis- covering functions, the fundamental units of logic in your game programs.

**Chapter 6: References: Tic-Tac-Toe.** You’ll learn how to share information with different parts of your programs in an efﬁcient and clear manner. You’ll also see a brief example of AI (*arti*ﬁ*cial intelligence*) and you’ll learn how to give a computer opponent a little bit of personality.

**Chapter 7: Pointers: Tic-Tac-Toe 2.0.** You’ll begin to discover some of the most low-level and powerful features of Cþþ, such as how to directly address and manipulate your computer’s memory.

**Chapter 8: Classes: Critter Caretaker.** You’ll learn how to create your own kinds of objects and deﬁne the ways they’ll interact with each other through object- oriented programming. In the process, you’ll create your very own critter to care for.

**Chapter 9: Advanced Classes and Dynamic Memory: Game Lobby.** You’ll expand on your direct connection with the computer and learn to acquire and free memory as your game programs require. You’ll also see the pitfalls of using this “dynamic” memory and how to avoid them.

**Chapter 10: Inheritance and Polymorphism: Blackjack.** You’ll learn how to deﬁne objects in terms of other objects. Then you’ll pull everything you’ve learned together into one big ﬁnal game. You’ll see how a sizeable project is designed and implemented by creating a version of the classic casino game of Blackjack (tacky green felt not included).

Introduction xxi

### Conventions Used in This Book

Throughout the book, I’ll throw in a few other tidbits. For example, I italicize any *new term* and explain what it means. I also use a number of special elements, including the following:

**Hint**

These are good ideas that will help you become a better game programmer.

**Trap**

These point out areas where it’s easy to make a mistake.

**Trick**

These suggest techniques and shortcuts that will make your life as a game programmer easier.

**Real World**

These are facts about the real world of game programming.

### Source Code for the Programs in this Book

All of the source code in this book is available online at [www.courseptr.com/](http://www.courseptr.com/downloads) [downloads.](http://www.courseptr.com/downloads) You can search for the book by ISBN (the book’s identiﬁcation number), which is 1435457420.

### A Word about Compilers

I might be getting a little ahead of myself here by talking about compilers, but the issue is important because a *compiler* is what translates the source code you write into a program that your computer can run. I recommend that you use Microsoft’s Visual Cþþ 2010 Express Edition, if you’re running Windows, since it includes a modern Cþþ compiler—and is free. Once you’ve installed the soft- ware, check out Appendix A in this book, “Creating Your First Cþþ Program,” which explains how to compile a Cþþ program using Visual Cþþ 2010 Express Edition. If you’re using another compiler or IDE, check its documentation.

*This page intentionally left blank*

|  |  |
| --- | --- |
|  |  |
| [chapter 1](#_bookmark0) | |

# [Types, Variables, and Standard I/O: Lost Fortune](#_bookmark0)

Game programming is demanding. It pushes both programmer and hardware to their limits. But it can also be extremely satisfying. In this chapter, you’ll be introduced to the fundamentals of Cþþ, the standard language for AAA game titles. Specifically, you’ll learn to:

n Display output in a console window

n Perform arithmetic computations

n Use variables to store, manipulate, and retrieve data

n Get user input

n Work with constants and enumerations

n Work with strings

### [Introducing Cþþ](#_bookmark0)

Cþþ is leveraged by millions of programmers around the world. It’s one of the most popular languages for writing computer applications—and *the* most popular language for writing big-budget computer games.

Created by Bjarne Stroustrup, Cþþ is a direct descendant of the C language. In fact, Cþþ retains almost all of C as a subset. However, Cþþ offers better ways to do things and some brand-new capabilities, too.

1

2 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

#### [Using Cþþ for Games](#_bookmark0)

There are a variety of reasons why game programmers choose Cþþ. Here are a few:

n **It’s fast.** Well-written Cþþ programs can be blazingly fast. One of Cþþ’s design goals is performance. And if you need to squeeze out even more performance from your programs, Cþþ allows you to use *assembly language*—the lowest-level, human-readable programming language—to communicate directly with the computer’s hardware.

n **It’s flexible.** Cþþ is a multi-paradigm language that supports different styles of programming, including *object-oriented programming*. Unlike some other modern languages, though, Cþþ doesn’t force one particular style on a programmer.

n **It’s well-supported.** Because of its long history in the game industry, there’s a large pool of assets available to the Cþþ game programmer, including graphics APIs and 2D, 3D, physics, and sound engines. All of this pre-exiting code can be leveraged by a Cþþ programmer to greatly speed up the process of writing a new game.

#### [Creating an Executable File](#_bookmark0)

The file that you run to launch a program—whether you’re talking about a game or a business application—is an *executable file*. There are several steps to creating an executable file from Cþþ *source code* (a collection of instructions in the Cþþ language). The process is illustrated in Figure 1.1.

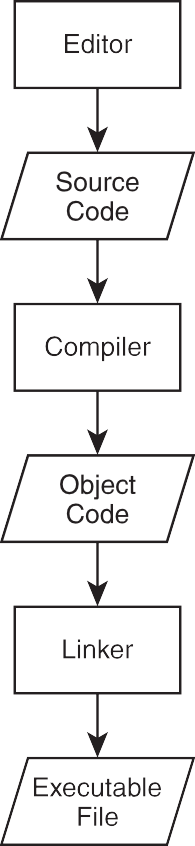
1. First, the programmer uses an *editor* to write the Cþþ source code, a file that usually has the extension .cpp. The editor is like a word

processor for programs; it allows a programmer to create, edit, and save source code.

1. After the programmer saves a source file, he or she invokes a Cþþ *compiler*—an application that reads source code and translates it into an *object file*. Object files usually have the extension .obj.
2. Next, a linker links the object file to any external files as necessary, and then creates the executable file, which generally ends with the extension

Introducing Cþþ 3

.exe. At this point, a user (or gamer) can run the program by launching the executable file.



**Figure 1.1**

The creation of an executable file from Cþþ source code.

**Hint**

The process I’ve described is the simple case. Creating a complex application in Cþþ often involves multiple source code files written by a programmer (or even a team of programmers).

To help automate this process, it’s common for a programmer to use an all-in- one tool for development, called an IDE (*Integrated Development Environment*). An IDE typically combines an editor, a compiler, and a linker, along with other

4 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

tools. A popular (and free) IDE for Windows is Microsoft’s Visual Cþþ Express Edition. You can find out more about this IDE (and download a copy) at [http://](http://www.microsoft.com/express/) [www.microsoft.com/express/.](http://www.microsoft.com/express/)

#### [Dealing with Errors](#_bookmark0)

When I described the process for creating an executable from Cþþ source, I left out one minor detail—errors. If to err is human, then programmers are the most human of us. Even the best programmers write code that generates errors the first (or fifth) time through. Programmers must fix the errors and start the entire process over. Here are the basic types of errors you’ll run into as you program in Cþþ:

n **Compile errors.** These occur during code compilation. As a result, an object file is not produced. These can be *syntax errors*, meaning that the compiler doesn’t understand something. They’re often caused by some- thing as simple as a typo. Compilers can issue warnings, too. Although you usually don’t have to heed the warnings, you should treat them as errors, fix them, and recompile.

n **Link errors.** These occur during the linking process and may indicate that something the program references externally can’t be found. These errors are usually solved by adjusting the offending reference and starting the compile/link process again.

n **Run-time errors.** These occur when the executable is run. If the program does something illegal, it can crash abruptly. But a more subtle form of run-time error, a *logical error*, can make the program simply behave in unintended ways. If you’ve ever played a game where a character walked on air (that is, a character who shouldn’t be able to walk on air), then you’ve seen a logical error in action.

**Real World**

Like other software creators, game companies work hard to produce bug-free products. Their last line of defense is the quality assurance personnel (the game testers). Game testers play games for a living, but their jobs are not as fun as you might think. Testers must play the same parts of a game over and over—perhaps hundreds of times—trying the unexpected and meticulously recording any anomalies. On top of monotonous work, the pay ain’t great either. But being a tester is a terrific way to get into a game company on the proverbial bottom rung.

Writing Your First Cþþ Program 5

#### [Understanding the ISO Standard](#_bookmark0)

The *ISO standard* for Cþþ is a definition of Cþþ that describes exactly how the language should work. It also defines a group of files, called the *standard library*, that contain building blocks for common programming tasks, such as *I/O*— getting input and displaying output. The standard library makes life easier for programmers and provides fundamental code to save them from reinventing the wheel. I’ll be using the standard library in all of the programs in this book.

**Hint**

The ISO standard is often called the *ANSI standard* or *ANSI/ISO standard*. These different names involve the acronyms of the various committees that have reviewed and established the standard. The most common way to refer to Cþþ code that conforms to the ISO standard is simply *Standard Cþþ*.

I used Microsoft’s Visual Cþþ 2010 Express Edition to develop the programs in this book. The compiler that’s a part of this IDE is pretty faithful to the ISO standard, so you should be able to compile, link, and run all of the programs using some other modern compiler as well. However, if you’re using Windows, I recommend using Visual Cþþ.

**Hint**

For step-by-step instructions on how to create, save, compile, and run the Game Over program using Microsoft Visual Cþþ 2010 Express Edition, check out Appendix A. If you’re using another compiler or IDE, check its documentation.

### [Writing Your First Cþþ Program](#_bookmark0)

Okay, enough theory. It’s time to get down to the nitty-gritty and write your first Cþþ program. Although it is simple, the following program shows you the basic anatomy of a program. It also demonstrates how to display text in a console window.

#### [Introducing the Game Over Program](#_bookmark0)

The classic first task a programmer tackles in a new language is the Hello World program, which displays Hello World on the screen. The Game Over program

6 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

puts a gaming twist on the classic and displays Game Over! instead. Figure 1.2 shows the program in action.



**Figure 1.2**

Your first Cþþ program displays the two most infamous words in computer gaming.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 1 folder; the filename is game\_over.cpp.

**Hi n t**

You can download all of the source code for the programs in this book by visiting [www.](http://www.courseptr.com/downloads) [courseptr.com/downloads](http://www.courseptr.com/downloads) and searching for this book. One way to search is by ISBN (the book's identification number), which is 1435457420.

// Game Over

// A first Cþþ program

#include <iostream> int main()

{

std::cout << "Game Over!" << std::endl; return 0;

}

Writing Your First Cþþ Program 7

#### [Commenting Code](#_bookmark0)

The first two lines of the program are comments.

// Game Over

// A first Cþþ program

Comments are completely ignored by the compiler; they’re meant for humans. They can help other programmers understand your intentions. But comments  can also help you. They can remind you how you accomplished something that might not be clear at first glance.

You can create a comment using two forward slashes in a row (//). Anything after this on the rest of the physical line is considered part of the comment. This means you can also include a comment after a piece of Cþþ code, on the same line.

**Hint**

You can also use what are called *C-style comments*, which can span multiple lines. All you have to do is start the comment with /\* and end it with \*/. Everything in between the two markers is part of the comment.

#### [Using Whitespace](#_bookmark0)

The next line in the program is a blank line. The compiler ignores blank lines. In fact, compilers ignore just about all *whitespace*—spaces, tabs, and newlines. Like comments, whitespace is just for us humans.

Judicious use of whitespace helps make programs clearer. For example, you can use blank lines to separate sections of code that belong together. I also use whitespace (a tab, to be precise) at the beginning of the two lines between the curly braces to set them off.

#### [Including Other Files](#_bookmark0)

The next line in the program is a preprocessor directive. You know this because the line begins with the # symbol.

#include <iostream>

The *preprocessor* runs before the compiler does its thing and substitutes text based on various directives. In this case, the line involves the #include directive, which tells the preprocessor to include the contents of another file.

8 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

I include the file iostream, which is part of the standard library, because it contains code to help me display output. I surround the filename with less than (<) and greater than (>) characters to tell the compiler to find the file where it keeps all the files that came with the compiler. A file that you include in your programs like this is called a *header file*.

#### [Defining the main() Function](#_bookmark1)

The next non-blank line is the header of a function called main().

int main()

A *function* is a group of programming code that can do some work and return a value. In this case, int indicates that the function will return an integer value. All function headers have a pair of parentheses after the function name.

All Cþþ programs must have a function called main(), which is the starting point of the program. The real action begins here.

The next line marks the beginning of the function.

{

And the very last line of the program marks the end of the function.

}

All functions are delimited by a pair of curly braces, and everything between them is part of the function. Code between two curly braces is called a *block* and is usually indented to show that it forms a unit. The block of code that makes up an entire function is called the *body* of the function.

#### [Displaying Text through the Standard Output](#_bookmark0)

The first line in the body of main() displays Game Over!, followed by a newline, in the console window.

std::cout << "Game Over!" << std::endl;

"Game Over!" is a *string*—a series of printable characters. Technically, it’s a *string literal*, meaning it’s literally the characters between the quotes.

cout is an object, defined in the file iostream, that’s used to send data to the standard output stream. In most programs (including this one), the standard output stream simply means the console window on the computer screen.

Writing Your First Cþþ Program 9

I use the *output operator* (<<) to send the string to cout. You can think of the output operator like a funnel; it takes whatever’s on the open side and funnels it to the pointy side. So the string is funneled to the standard output—the screen.

I use std to prefix cout to tell the compiler that I mean cout from the standard library. std is a *namespace*. You can think of a namespace like an area code of a phone number—it identifies the group to which something belongs. You prefix a namespace using the *scope resolution operator* (::).

Finally, I send std::endl to the standard output. endl is defined in iostream and is also an object in the std namespace. Sending endl to the standard output acts like pressing the Enter key in the console window. In fact, if I were to send another string to the console window, it would appear on the next line.

I understand this might be a lot to take in, so check out Figure 1.3 for a visual representation of the relationship between all of the elements I’ve just described.





**Figure 1.3**

An implementation of Standard Cþþ includes a set of files called the standard library, which includes the file iostream, which defines various things, including the object cout.

#### [Terminating Statements](#_bookmark0)

You’ll notice that the first line of the function ends with a semicolon (;). That’s because the line is a *statement*—the basic unit controlling the execution flow. All of your statements must end with a semicolon—otherwise, your compiler will complain with an error message and your program won’t compile.

10 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

#### [Returning a Value from main()](#_bookmark0)

The last statement in the function returns 0 to the operating system.

return 0;

Returning 0 from main() is a way to indicate that the program ended without a problem. The operating system doesn’t have to do anything with the return value. In general, you can simply return 0 like I did here.

**Trick**

When you run the Game Over program, you might only see a console window appear and disappear just as quickly. That’s because Cþþ is so fast that it opens a console window, displays Game Over!, and closes the window all in a split second. However, in Windows, you can create a batch file that runs your console program and pauses, keeping the console window open so you can see the results of your program. Since the compiled program is named game\_over.exe, you can simply create a batch file comprised of the two lines

game\_over.exe pause

To create a batch file:

1. Open a text editor like Notepad (not Word or WordPad).
2. Type your text.
3. Save the file with a .bat extension, such as game\_over.bat.

Finally, run the batch file by double-clicking its icon. You should see the results of the program since the batch file keeps the console window open.

### [Working with the std Namespace](#_bookmark0)

Because it’s so common to use elements from the std namespace, I’ll show you two different methods for directly accessing these elements. This will save you the effort of using the std:: prefix all the time.

#### [Introducing the Game Over 2.0 Program](#_bookmark0)

The Game Over 2.0 program produces the exact results of the original Game Over program, illustrated in Figure 1.2. But there’s a difference in the way elements from the std namespace are accessed. You can download the code for

Working with the std Namespace 11

this program from the Course Technology website [(www.courseptr.com/down-](http://www.courseptr.com/downloads) [loads).](http://www.courseptr.com/downloads) The program is in the Chapter 1 folder; the filename is game\_over2.cpp.

// Game Over 2.0

// Demonstrates a using directive

#include <iostream> using namespace std;

int main()

{

cout << "Game Over!" << endl; return 0;

}

#### [Employing a using Directive](#_bookmark0)

The program starts in the same way. I use two opening comments and then include iostream for output. But next, I have a new type of statement.

using namespace std;

This using directive gives me direct access to elements of the std namespace. Again, if a namespace is like an area code, then this line says that all of the elements in the std namespace should be like local phone numbers to me now. That is, I don’t have to use their area code (the std:: prefix) to access them.

I can use cout and endl, without any kind of prefix. This might not seem like a big deal to you now, but when you have dozens or even hundreds of references to these objects, you’ll thank me.

#### [Introducing the Game Over 3.0 Program](#_bookmark0)

Okay, there’s another way to accomplish what I did in Game Over 2.0: set up the file so that I don’t have to explicitly use the std:: prefix to access cout and endl. And that’s exactly what I’m going to show you in the Game Over 3.0 program, which displays the same text as its predecessors. You can download the code for this program from the Course Technology website [(www.courseptr.com/down-](http://www.courseptr.com/downloads) [loads).](http://www.courseptr.com/downloads) The program is in the Chapter 1 folder; the filename is game\_over3.cpp.

// Game Over 3.0

// Demonstrates using declarations

12 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

#include <iostream> using std::cout; using std::endl;

int main()

{

cout << "Game Over!" << endl; return 0;

}

#### [Employing using Declarations](#_bookmark0)

In this version, I write two using declarations.

using std::cout; using std::endl;

By declaring exactly which elements from the std namespace I want local to my program, I’m able to access them directly, just as in Game Over 2.0. Although it requires more typing than a using directive, the advantage of this technique is that it clearly spells out those elements I plan to use. Plus, it doesn’t make local a bunch of other elements that I have no intention of using.

#### [Understanding When to Employ using](#_bookmark0)

Okay, you’ve seen two ways to make elements from a namespace local to your program. But which is the best technique?

A language purist would say you shouldn’t employ either version of using and that you should always prefix each and every element from a namespace with its identifier. In my opinion, that’s like calling your best friend by his first and last name all the time. It just seems a little too formal.

If you hate typing, you can employ the using directive. A decent compromise is to employ using declarations. In this book, I’ll employ the using directive most of the time for brevity’s sake.

**Real World**

I’ve laid out a few different options for working with namespaces. I’ve also tried to explain the advantages of each so you can decide which way to go in your own programs. Ultimately, though, the decision may be out of your hands. When you’re working on a project, whether it’s in the classroom or in the professional world, you’ll probably receive coding standards created by the

Using Arithmetic Operators 13

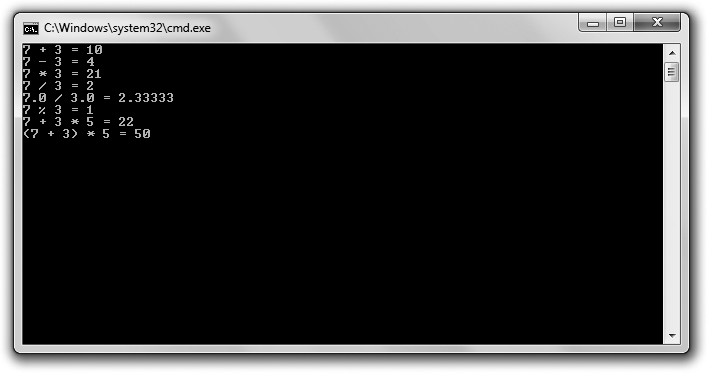
person in charge. Regardless of your personal tastes, it’s always best to listen to those who hand out grades or paychecks.

### [Using Arithmetic Operators](#_bookmark2)

Whether you’re tallying up the number of enemies killed or decreasing a player’s health level, you need your programs to do some math. As with other languages, Cþþ has built-in arithmetic operators.

#### [Introducing the Expensive Calculator Program](#_bookmark2)

Most serious computer gamers invest heavily in a bleeding-edge, high-powered gaming rig. This next program, Expensive Calculator, can turn that monster of a machine into a simple calculator. The program demonstrates built-in arithmetic operators. Figure 1.4 shows off the results.



**Figure 1.4**

Cþþ can add, subtract, multiply, divide, and even calculate a remainder.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 1 folder; the filename is expensive\_calculator.cpp.

// Expensive Calculator

// Demonstrates built-in arithmetic operators

14 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

#include <iostream> using namespace std;

int main()

{

cout << "7 + 3 = " << 7 + 3 << endl;

cout << "7 - 3 = " << 7 - 3 << endl;

cout << "7 \* 3 = " << 7 \* 3 << endl;

cout << "7 / 3 = " << 7 / 3 << endl;

cout << "7.0 / 3.0 = " << 7.0 / 3.0 << endl;

cout << "7 % 3 = " << 7 % 3 << endl;

cout << "7 + 3 \* 5 = " << 7 + 3 \* 5 << endl;

cout << "(7 + 3) \* 5 = " << (7 + 3) \* 5 << endl;

return 0;

}

#### [Adding, Subtracting, and Multiplying](#_bookmark2)

I use the built-in arithmetic operators for addition (the plus sign, þ), subtraction (the minus sign, -), and multiplication (an asterisk, \*). The results depicted in Figure 1.4 are just what you’d expect.

Each arithmetic operator is part of an *expression*—something that evaluates to a single value. So, for example, the expression 7 þ 3 evaluates to 10, and that’s what gets sent to cout.

#### [Understanding Integer and Floating Point Division](#_bookmark2)

The symbol for division is the forward slash (/), so that’s what I use in the next line of code. However, the output might surprise you. According to Cþþ (and that expensive gaming rig), 7 divided by 3 is 2. What’s going on? Well, the result of any arithmetic calculation involving only *integers* (numbers without fractional parts) is always another integer. And since 7 and 3 are both integers, the result must be an integer. The fractional part of the result is thrown away.

To get a result that includes a fractional part, at least one of the values needs to be a *floating point* (a number with a fractional part). I demonstrate this in the

Using Arithmetic Operators 15

next line with the expression 7.0 / 3.0. This time the result is a more accurate

2.33333.

**Trap**

You might notice that while the result of 7.0 / 3.0 (2.33333) includes a fractional part, it is still truncated. (The true result would stretch out 3s after the decimal point forever.) It’s important to know that computers generally store only a limited number of significant digits for floating point numbers. However, Cþþ offers categories of floating point numbers to meet the most demanding needs—even those of computationally intensive 3D games.

#### [Using the Modulus Operator](#_bookmark2)

In the next statement, I use an operator that might be unfamiliar to you—the modulus operator (%). The modulus operator returns the remainder of integer division. In this case, 7 % 3 produces the remainder of 7 / 3, which is 1.

#### [Understanding Order of Operations](#_bookmark2)

Just as in algebra, arithmetic expressions in Cþþ are evaluated from left to right. But some operators have a higher precedence than others and are evaluated first, regardless of position. Multiplication, division, and modulus have equal prece- dence, which is higher than the precedence level that addition and subtraction share.

The next line of code provides an example to help drive this home. Because multiplication has higher precedence than addition, you calculate the results of the multiplication first. So the expression 7 þ 3 \* 5 is equivalent to 7 þ 15, which evaluates to 22.

If you want an operation with lower precedence to occur first, you can use parentheses, which have higher precedence than any arithmetic operator. So in the next statement, the expression (7 þ 3) \* 5 is equivalent to 10 \* 5, which evaluates to 50.

**Hint**

For a list of Cþþ operators and their precedence levels, see Appendix B.

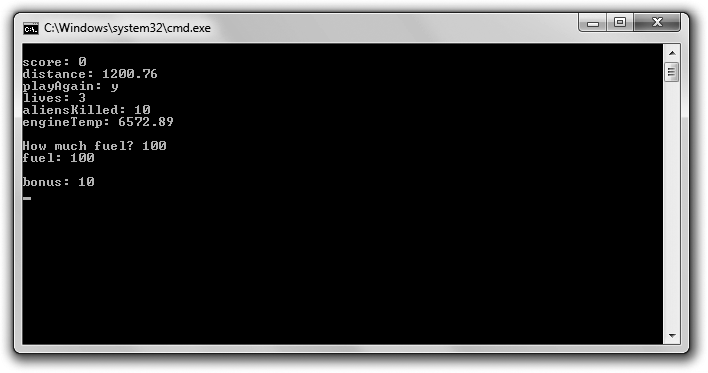
16 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

### [Declaring and Initializing Variables](#_bookmark2)

A *variable* represents a particular piece of your computer’s memory that has been set aside for you to use to store, retrieve, and manipulate data. So if you wanted to keep track of a player’s score, you could create a variable for it, then you could retrieve the score to display it. You could also update the score when the player blasts an alien enemy from the sky.

#### [Introducing the Game Stats Program](#_bookmark2)

The Game Stats program displays information that you might want to keep track of in a space shooter game, such as a player’s score, the number of enemies the player has destroyed, and whether the player has his shields up. The program uses a group of variables to accomplish all of this. Figure 1.5 illustrates the program.



**Figure 1.5**

Each game stat is stored in a variable.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 1 folder; the filename is game\_stats.cpp.

// Game Stats

// Demonstrates declaring and initializing variables

Declaring and Initializing Variables 17

#include <iostream> using namespace std;

int main()

{

int score; double distance; char playAgain; bool shieldsUp;

short lives, aliensKilled; score = 0;

distance = 1200.76;

playAgain = ’y’; shieldsUp = true; lives = 3;

aliensKilled = 10;

double engineTemp = 6572.89;

cout << "\nscore: " << score << endl; cout << "distance: " << distance << endl; cout << "playAgain: " << playAgain << endl;

//skipping shieldsUp since you don’t generally print Boolean values cout << "lives: " << lives << endl;

cout << "aliensKilled: "<< aliensKilled << endl; cout << "engineTemp: " << engineTemp << endl;

int fuel;

cout << "\nHow much fuel? "; cin >> fuel;

cout << "fuel: " << fuel << endl;

typedef unsigned short int ushort; ushort bonus = 10;

cout << "\nbonus: " << bonus << endl;

return 0;

}

18 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

#### [Understanding Fundamental Types](#_bookmark2)

Every variable you create has a *type*, which represents the kind of information you can store in the variable. It tells your compiler how much memory to set aside for the variable and it defines exactly what you can legally do with the variable.

*Fundamental types*—those built into the language—include bool for Boolean values (true or false), char for single character values, int for integers, float for single-precision floating point numbers, and double for double-precision floating point numbers.

#### [Understanding Type Modifiers](#_bookmark2)

You can use modifiers to alter a type. short is a modifier that can reduce the total number of values a variable can hold. long is a modifier that can increase the total number of values a variable can hold. short may decrease the storage space required for a variable while long may increase it. short and long can modify int. long can also modify double.

signed and unsigned are modifiers that work only with integer types. signed means that a variable can store both positive and negative values, while unsigned means that a variable can store only positive values. Neither signed nor unsigned change the total number of values a variable can hold; they only change the range of values. signed is the default for integer types.

Okay, confused with all of your type options? Well, don’t be. Table 1.1 summarizes commonly used types with some modifiers thrown in. The table also provides a range of values for each.

**Trap**

The range of values listed is based on my compiler. Yours might be different. Check your compiler’s documentation.

**Hi n t**

For brevity’s sake, short int can be written as just short and long int can be written as just

long.

Declaring and Initializing Variables 19

-32,768 to 32,767

0 to 65,535

-2,147,483,648 to 2,147,483,647

0 to 4,294,967,295

-2,147,483,648 to 2,147,483,647

0 to 4,294,967,295

3.4E þ/- 38 (seven significant digits) 1.7E þ/- 308 (15 significant digits)

1.7E þ/- 308 (15 significant digits)

256 character values

true or false

short int

unsigned short int int

unsigned int long int

unsigned long int float

double

long double char

bool

**Table 1.1** Commonly Used Types

**Type Values**

#### [Declaring Variables](#_bookmark2)

All right, now that you’ve got a basic understanding of types, it’s time to get back to the program. One of the first things I do is *declare* a variable (request that it be created) with the line:

int score;

In this code, I declare a variable of type int, which I name score. You use a variable name to access the variable. You can see that to declare a variable you specify its type followed by a name of your choosing. Because the declaration is a statement, it must end with a semicolon.

I declare three more variables of yet three more types in the next three lines. distance is a variable of type double. playAgain is a variable of type char. And shieldsUp is a variable of type bool.

Games (and all major applications) usually require lots of variables. Fortunately, Cþþ allows you to declare multiple variables of the same type in a single statement. That’s just what I do next in the line.

short lives, aliensKilled;

This line establishes two short variables—lives and aliensKilled.

20 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

Even though I’ve defined a bunch of variables at the top of my main() function, you don’t have to declare all of your variables in one place. As you’ll see later in the program, I often define a new variable just before I use it.

#### [Naming Variables](#_bookmark2)

To declare a variable, you must provide a name, known as an *identifier*. There are only a few rules you have to follow to create a legal identifier.

n An identifier can contain only numbers, letters, and underscores.

n An identifier can’t start with a number.

n An identifier can’t be a Cþþ keyword.

A *keyword* is a special word that Cþþ reserves for its own use. There aren’t many, but to see a full list, check out Appendix C.

In addition to the rules for creating *legal* variable names, following are some guidelines for creating *good* variable names.

n **Choose descriptive names.** Variable names should be clear to another programmer. For example, use score instead of s. (One exception to this rule involves variables used for a brief period. In that case, single-letter variable names, such as x, are fine.)

n **Be consistent.** There are different schools of thought about how to write multiword variable names. Is it high\_score or highScore? In this book, I use the second style, where the initial letter of the second word (and any other words) is capitalized, known as *camel case*. But as long as you’re consistent, it’s not important which method you use.

n **Follow the traditions of the language.** Some naming conventions are just traditions. For example, in most languages (Cþþ included) variable names start with a lowercase letter. Another tradition is to avoid using an underscore as the first character of your variable names. Names that begin with an underscore can have special meaning.

n **Keep the length in check.** Even though playerTwoBonusForRoundOne is descriptive, it can make code hard to read. Plus, long names increase the risk of a typo. As a guideline, try to limit your variable names to fewer

Declaring and Initializing Variables 21

than 15 characters. Ultimately, though, your compiler sets an actual upper limit.

**Trick**

*Self-documenting* code is written in such a way that it’s easy to understand what is happening in the program independent of any comments. Choosing good variable names is an excellent step toward this kind of code.

#### [Assigning Values to Variables](#_bookmark2)

In the next group of statements, I assign values to the six variables I declared. I’ll go through a few assignments and talk a little about each variable type.

***Assigning Values to Integer Variables***

In the following assignment statement I assign the value of 0 to score.

score = 0;

Now score stores 0.

You assign a value to a variable by writing the variable name followed by the assignment operator (=) followed by an expression. (Yes, technically 0 is an expression, which evaluates to, well, 0.)

***Assigning Values to Floating Point Variables***

In the statement I assign distance the value 1200.76.

distance = 1200.76;

Because distance is of type double, I can use it to store a number with a fractional part, which is just what I do.

***Assigning Values to Character Variables***

In the following statement I assign playAgain the single-character value ’y’.

playAgain = ’y’;

As I did here, you can assign a character to a variable of type char by surrounding the character with single quotes.

22 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

Variables of type char can store the 128 ASCII character values (assuming that your system uses the ASCII character set). *ASCII*, short for *American Standard Code for Information Interchange*, is a code for representing characters. To see a complete ASCII listing, check out Appendix D.

***Assigning Values to Boolean Variables***

In the following statement I assign shieldsUp the value true.

shieldsUp = true;

In my program, this means that the player’s shields are up.

shieldsUp is a bool variable, which means it’s a Boolean variable. As such, it can represent either true or false. Although intriguing, you’ll have to wait until Chapter 2, “Truth, Branching, and the Game Loop: Guess My Number,” to learn more about this kind of variable.

#### [Initializing Variables](#_bookmark2)

You can both declare and assign a value to variables in a single initialization statement. That’s exactly what I do next.

double engineTemp = 6572.89;

This line creates a variable of type double named engineTemp, which stores the value 6572.89.

Just as you can declare multiple variables in one statement, you can initialize more than one variable in a statement. You can even declare and initialize different variables in a single statement. Mix and match as you choose!

**Hi n t**

Although you can declare a variable without assigning it a value, it’s best to initialize a new variable with a starting value whenever you can. This makes your code clearer, plus it eliminates the chance of accessing an uninitialized variable, which may contain any value.

#### [Displaying Variable Values](#_bookmark2)

To display the value of a variable of one of the fundamental types, just send it to cout. That’s what I do next in the program. Note that I don’t try to display shieldsUp because you don’t normally display bool values.

Declaring and Initializing Variables 23

**Trick**

In the first statement of this section I use what’s called an *escape sequence*—a pair of characters that begins with a backslash (\), which represents special printable characters.

cout << "\nscore: " << score << endl;

The escape sequence I used is \n, which represents a newline. When sent to cout as part of a string, it’s like pressing the Enter key in the console window. Another useful escape sequence is \t, which acts as a tab.

There are other escape sequences at your disposal. For a list of escape sequences, see Appendix E.

#### [Getting User Input](#_bookmark2)

Another way to assign a value to a variable is through user input. So next, I assign the value of a new variable, fuel, based on what the user enters. To do so I use the following line:

cin >> fuel;

Just like cout, cin is an object defined in iostream, which lives in the std namespace. To store a value in the variable, I use cin followed by >> (the extraction operator), followed by the variable name. You can use cin and the extraction operator to get user input into variables of other fundamental types too. To prove that everything worked, I display fuel to the user.

#### [Defining New Names for Types](#_bookmark2)

You can define a new name for an existing type. In fact, that’s what I do next in the line:

typedef unsigned short int ushort;

This code defines the identifier ushort as another name for the type unsigned short int. To define new names for existing types, use typedef followed by the current type, followed by the new name. typedef is often used to create shorter names for types with long names.

You can use your new type name just like the original type. I initialize a ushort variable (which is really just an unsigned short int) named bonus and display its value.

24 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

#### [Understanding Which Types to Use](#_bookmark2)

You have many choices when it comes to the fundamental types. So how do you know which type to use? Well, if you need an integer type, you’re probably best off using int. That’s because int is generally implemented so that it occupies an amount of memory that is most efficiently handled by the computer. If you need to represent integer values greater than the maximum int or values that will never be negative, feel free to use an unsigned int.

If you’re tight on memory, you can use a type that requires less storage. However, on most computers, memory shouldn’t be much of an issue. (Pro- gramming on game consoles or mobile devices is another story.)

Finally, if you need a floating point number, you’re probably best off using float, which again is likely to be implemented so that it occupies an amount of memory that is most efficiently handled by the computer.

### Performing Arithmetic Operations [with Variables](#_bookmark2)

Once you have variables with values, you’ll want to change their values during the course of your game. You might want to add a bonus to a player’s score for defeating a boss, increasing the score. Or you might want to decrease the oxygen level in an airlock. By using operators you’ve already met (along with some new ones), you can accomplish all of this.

#### [Introducing the Game Stats 2.0 Program](#_bookmark2)

The Game Stats 2.0 program manipulates variables that represent game stats and displays the results. Figure 1.6 shows the program in action.

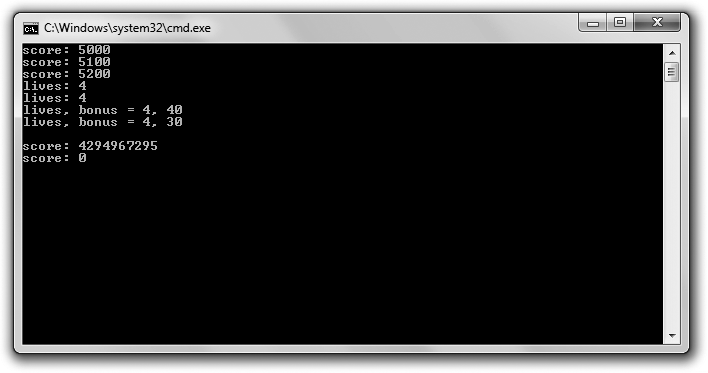
You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 1 folder; the filename is game\_stats2.cpp.

// Game Stats 2.0

// Demonstrates arithmetic operations with variables

#include <iostream> using namespace std;

Performing Arithmetic Operations with Variables 25



**Figure 1.6**

Each variable is altered in a different way.

int main()

{

unsigned int score = 5000;

cout << "score: " << score << endl;

//altering the value of a variable score = score + 100;

cout << "score: " << score << endl;

//combined assignment operator score += 100;

cout << "score: " << score << endl;

//increment operators int lives = 3; þþlives;

cout << "lives: " << lives << endl;

lives = 3; livesþþ;

cout << "lives: " << lives << endl;

26 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

lives = 3;

int bonus = þþlives \* 10;

cout << "lives, bonus = " << lives << ", " << bonus << endl;

lives = 3;

bonus = livesþþ \* 10;

cout << "lives, bonus = " << lives << ", " << bonus << endl;

//integer wrap around score = 4294967295;

cout << "\nscore: " << score << endl;

þþscore;

cout << "score: " << score << endl;

return 0;

}

**Trap**

When you compile this program, you may get a warning similar to, “[Warning] this decimal constant is unsigned.” Fortunately, the warning does not stop the program from compiling and being run. The warning is the result of something called integer wrap around that you’ll probably want to avoid in your own programs; however, the wrap around is intentional in this program to show the results of the event. You’ll learn about integer wrap around in the discussion of this program, in the section “Dealing with Integer Wrap Around.”

#### [Altering the Value of a Variable](#_bookmark2)

After I create a variable to hold the player’s score and display it, I alter the score by increasing it by 100.

score = score + 100;

This assignment statement says to take the current value of score, add 100, and assign the result back to score. In effect, the line increases the value of score by 100.

#### [Using Combined Assignment Operators](#_bookmark2)

There’s an even shorter version of the preceding line, which I use next.

score += 100;

This statement produces the same results as score = score þ 100;. The þ=

operator is called a *combined assignment operator* because it combines an

Performing Arithmetic Operations with Variables 27

arithmetic operation (addition, in this case) with assignment. This operator is shorthand for saying “add whatever’s on the right to what’s on the left and assign the result back to what’s on the left.”

There are versions of the combined assignment operator for all of the arithmetic operators you’ve met. To see a list, check out Table 1.2.

|  |  |  |
| --- | --- | --- |
| **Table 1.2** Combined Assignment Operators | | |
| **Operator** | **Example** | **Equivalent To** |
| += | x += 5; | x = x + 5; |
| -= | x -= 5; | x = x - 5; |
| \*= | x \*= 5; | x = x \* 5; |
| /= | x /= 5; | x = x / 5; |
| %= | x %= 5; | x = x % 5; |

#### [Using Increment and Decrement Operators](#_bookmark2)

Next, I use the *increment operator* (þþ) which increases the value of a variable by one. I use the operator to increase the value of lives twice. First I use it in the following line:

þþlives;

Then I use it again in the following line:

livesþþ;

Each line has the same net effect; it increments lives from 3 to 4.

As you can see, you can place the operator before or after the variable you’re incrementing. When you place the operator before the variable, the operator is called the *prefix increment operator*; when you place it after the variable, it’s called the *postfix increment operator*.

At this point, you might be thinking that there’s no difference between the postfix and prefix versions, but you’d be wrong. In a situation where you only increment a single variable (as you just saw), both operators produce the same final result. But in a more complex expression, the results can be different.

To demonstrate this important difference, I perform a calculation that would be appropriate for the end of a game level. I calculate a bonus based on the number

28 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

of lives a player has, and I also increment the number of lives. However, I perform this calculation in two different ways. The first time, I use the prefix increment operator.

int bonus = þþlives \* 10;

The prefix increment operator increments a variable *before* the evaluation of a larger expression involving the variable. þþlives \* 10 is evaluated by first incrementing lives, and then multiplying that result by 10. Therefore, the code is equivalent to 4\* 10, which is 40, of course. This means that now lives is 4 and bonus is 40.

After setting lives back to 3, I calculate bonus again, this time using the postfix increment operator.

bonus = livesþþ \* 10;

The postfix increment operator increments a variable *after* the evaluation of a larger expression involving the variable. livesþþ \* 10 is evaluated by multi- plying the current value of lives by 10. Therefore, the code is equivalent to 3 \* 10, which is 30, of course. Then, after this calculation, lives is incremented. After the line is executed, lives is 4 and bonus is 30.

Cþþ also defines the *decrement operator*, –. It works just like the increment operator, except it decrements a variable. It comes in the two flavors (prefix and postfix) as well.

#### [Dealing with Integer Wrap Around](#_bookmark2)

What happens when you increase an integer variable beyond its maximum value? It turns out you don’t generate an error. Instead, the value “wraps around” to the type’s minimum value. Next up, I demonstrate this phenomenon. First I assign score the largest value it can hold.

score = 4294967295;

Then I increment the variable.

þþscore;

As a result, score becomes 0 because the value wrapped around, much like a car odometer does when it goes beyond its maximum value (see Figure 1.7).

Decrementing an integer variable beyond its minimum value “wraps it around” to its maximum.

Working with Constants 29

**Figure 1.7**

A way to visualize an unsigned int variable “wrapping around” from its maximum value to its minimum.

**Hint**

Make sure to pick an integer type that has a large enough range for its intended use.

### [Working with Constants](#_bookmark2)

A *constant* is an unchangeable value that you name. Constants are useful if you have an unchanging value that comes up frequently in your program. For example, if you were writing a space shooter in which each alien blasted out of the sky is worth 150 points, you could define a constant named ALIEN\_POINTS that is equal to 150. Then, any time you need the value of an alien, you could use ALIEN\_POINTS instead of the literal 150.

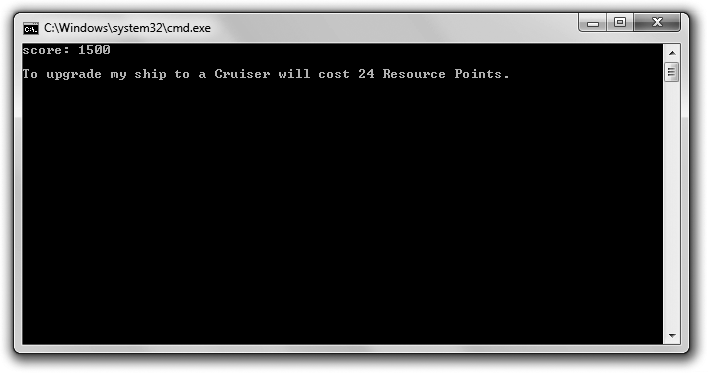
Constants provide two important benefits. First, they make programs clearer. As soon as you see ALIEN\_POINTS, you know what it means. If you were to look at some code and see 150, you might not know what the value represents. Second, constants make changes easy. For example, suppose you do some playtesting with your game and you decide that each alien should really be worth 250 points. With constants, all you’d have to do is change the initialization of ALIEN\_POINTS in your program. Without constants, you’d have to hunt down every occurrence of 150 and change it to 250.

#### [Introducing the Game Stats 3.0 Program](#_bookmark2)

The Game Stats 3.0 program uses constants to represent values. First the program calculates a player’s score, and then it calculates the upgrade cost of a unit in a strategy game. Figure 1.8 shows the results.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 1 folder; the filename is game\_stats3.cpp.

30 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune



**Figure 1.8**

Each calculation involves a constant, making the code behind the scenes clearer.

// Game Stats 3.0

// Demonstrates constants

#include <iostream> using namespace std;

int main()

{

const int ALIEN\_POINTS = 150; int aliensKilled = 10;

int score = aliensKilled \* ALIEN\_POINTS; cout << "score: " << score << endl;

enum difficulty {NOVICE, EASY, NORMAL, HARD, UNBEATABLE};

difficulty myDifficulty = EASY;

enum shipCost {FIGHTER\_COST = 25, BOMBER\_COST, CRUISER\_COST = 50};

shipCost myShipCost = BOMBER\_COST;

cout << "\nTo upgrade my ship to a Cruiser will cost "

<< (CRUISER\_COST - myShipCost) << " Resource Points.\n";

return 0;

}

Working with Constants 31

#### [Using Constants](#_bookmark2)

I define a constant, ALIEN\_POINTS, to represent the point value of an alien.

const int ALIEN\_POINTS = 150;

I simply use the keyword const to modify the definition. Now I can use ALIEN\_POINTS just like any integer literal. Also, notice that the name I chose for the constant is in all capital letters. This is just a convention, but it’s a common one. An identifier in all caps tells a programmer that it represents a constant value.

Next I put the constant to use in the following line:

int score = aliensKilled \* ALIEN\_POINTS;

I calculate a player’s score by multiplying the number of aliens killed by the point value of an alien. Using a constant here makes the line of code quite clear.

**Trap**

You can’t assign a new value to a constant. If you try, you’ll generate a compile error.

#### [Using Enumerations](#_bookmark2)

An *enumeration* is a set of unsigned int constants, called *enumerators*. Usually the enumerators are related and have a particular order. Here’s an example of an enumeration:

enum difficulty {NOVICE, EASY, NORMAL, HARD, UNBEATABLE};

This defines an enumeration named difficulty. By default, the value of enumerators begins at zero and increases by one. So NOVICE is 0, EASY is 1, NORMAL is 2, HARD is 3, and UNBEATABLE is 4. To define an enumeration of your own, use the keyword enum followed by an identifier, followed by a list of enumerators between curly braces.

Next I create a variable of this new enumeration type.

difficulty myDifficulty = EASY;

The variable myDifficulty is set to EASY (which is equal to 1). myDifficulty is of type difficulty, so it can only hold one of the values defined in the enumeration. That means myDifficulty can only be assigned NOVICE, EASY, NORMAL, HARD, UNBEATABLE, 0, 1, 2, 3, or 4.

32 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

Next I define another enumeration.

enum shipCost {FIGHTER\_COST = 25, BOMBER\_COST, CRUISER\_COST = 50};

This line of code defines the enumeration shipCost, which represents the cost in Resource Points for three kinds of ships in a strategy game. In it, I assign specific integer values to some of the enumerators. The numbers represent the Resource Point value of each ship. You can assign values to the enumerators if you want. Any enumerators that are not assigned values get the value of the previous enumerator plus one. Because I didn’t assign a value to BOMBER\_COST, it’s initialized to 26.

Next I define a variable of this new enumeration type.

shipCost myShipCost = BOMBER\_COST;

Then I demonstrate how you can use enumerators in arithmetic calculations.

(CRUISER\_COST - myShipCost)

This piece of code calculates the cost of upgrading a Bomber to a Cruiser. The calculation is the same as 50 - 26, which evaluates to 24.

### [Introducing Lost Fortune](#_bookmark2)

The final project for this chapter, Lost Fortune, is a personalized adventure game in which the player enters a few pieces of information, which the computer uses to enhance a basic adventure story. Figure 1.9 shows a sample run.

Instead of presenting all the code at once, I’ll go through it one section at a time. You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 1 folder; the filename is lost\_fortune.cpp.

#### [Setting Up the Program](#_bookmark2)

First I create some initial comments, include two necessary files, and write a few

using directives.

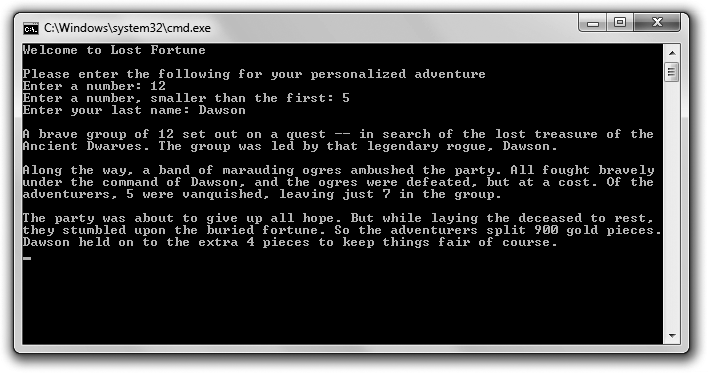
// Lost Fortune

// A personalized adventure

#include <iostream> #include <string>

Introducing Lost Fortune 33

using std::cout; using std::cin; using std::endl; using std::string;



**Figure 1.9**

The story incorporates details provided by the player.

I include the file string, part of the standard library, so I can use a string object to access a string through a variable. There’s a lot more to string objects, but I’m going to keep you in suspense. You’ll learn more about them in Chapter 3, “For Loops, Strings, and Arrays: Word Jumble.”

Also, I employ using directives to spell out the objects in the std namespace that I plan to access. As a result, you can clearly see that string is in namespace std.

#### [Getting Information from the Player](#_bookmark2)

Next I get some information from the player.

int main()

{

const int GOLD\_PIECES = 900;

int adventurers, killed, survivors; string leader;

34 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

//get the information

cout << "Welcome to Lost Fortune\n\n";

cout << "Please enter the following for your personalized adventure\n";

cout << "Enter a number: "; cin >> adventurers;

cout << "Enter a number, smaller than the first: "; cin >> killed;

survivors = adventurers - killed;

cout << "Enter your last name: "; cin >> leader;

GOLD\_PIECES is a constant that stores the number of gold pieces in the fortune the adventurers seek. adventurers stores the number of adventurers on the quest. killed stores the number that are killed on the journey. I calculate survivors for the number of adventurers that remain. Finally, I get the player’s last name, which I’ll be able to access through leader.

**Trap**

This simple use of cin to get a string from the user only works with strings that have no whitespace in them (such as tabs or spaces). There are ways to compensate for this, but that really requires a discussion of something called *streams*, which is beyond the scope of this chapter. So, use cin in this way, but be aware of its limitations.

#### [Telling the Story](#_bookmark2)

Next I use the variables to tell the story.

//tell the story

cout << "\nA brave group of " << adventurers << " set out on a quest "; cout << "-– in search of the lost treasure of the Ancient Dwarves. "; cout << "The group was led by that legendary rogue, " << leader << ".\n";

cout << "\nAlong the way, a band of marauding ogres ambushed the party. "; cout << "All fought bravely under the command of " << leader;

cout << ", and the ogres were defeated, but at a cost. ";

cout << "Of the adventurers, " << killed << " were vanquished, "; cout << "leaving just " << survivors << " in the group.\n";

Summary 35

cout << "\nThe party was about to give up all hope. "; cout << "But while laying the deceased to rest, "; cout << "they stumbled upon the buried fortune. ";

cout << "So the adventurers split " << GOLD\_PIECES << " gold pieces."; cout << leader << " held on to the extra " << (GOLD\_PIECES % survivors); cout << " pieces to keep things fair of course.\n";

return 0;

}

The code and thrilling narrative are pretty clear. I will point out one thing, though. To calculate the number of gold pieces that the leader keeps, I use the modulus operator in the expression GOLD\_PIECES % survivors. The expression evaluates to the remainder of GOLD\_PIECES / survivors, which is the number of gold pieces that would be left after evenly dividing the stash among all of the surviving adventurers.

### [Summary](#_bookmark2)

In this chapter, you should have learned the following concepts:

n Cþþ is the primary language used in AAA game programming.

n A program is a series of Cþþ statements.

n The basic lifecycle of a Cþþ program is idea, plan, source code, object file, executable.

n Programming errors tend to fall into three categories—compile errors, link errors, and run-time errors.

n A function is a group of programming statements that can do some work and return a value.

n Every program must contain a main() function, which is the starting point of the program.

n The #include directive tells the preprocessor to include another file in the current one.

n The standard library is a set of files that you can include in your program files to handle basic functions like input and output.

36 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

n iostream, which is part of the standard library, is a file that contains code to help with standard input and output.

n The std namespace includes elements from the standard library. To access an element from the namespace, you need to prefix the element with std:: or employ using.

n cout is an object, defined in the file iostream, that’s used to send data to the standard output stream (generally the computer screen).

n cin is an object, defined in the file iostream, that’s used to get data from the standard input stream (generally the keyboard).

n Cþþ has built-in arithmetic operators, such as the familiar addition, subtraction, multiplication, and division—and even the unfamiliar modulus.

n Cþþ defines fundamental types for Boolean, single-character, integer, and floating point values.

n The Cþþ standard library provides a type of object (string) for strings.

n You can use typedef to create a new name for an existing type.

n A constant is a name for an unchangeable value.

n An enumeration is a sequence of unsigned int constants.

### [Questions and Answers](#_bookmark2)

Q: Why do game companies use Cþþ?

A: Cþþ combines speed, low-level hardware access, and high-level constructs better than just about any other language. In addition, most game companies have a lot invested in Cþþ resources (both in reusable code and programmer experience).

Q: How is Cþþ different than C?

A: Cþþ is the next iteration of the C programming language. To gain accept- ance, Cþþ essentially retained all of C. However, Cþþ defines new ways to do things that can replace some of the traditional C mechanisms. In addition, Cþþ adds the ability to write object-oriented programs.

Questions and Answers 37

Q: How should I use comments?

A: To explain code that is unusual or unclear. You should not comment the obvious.

Q: What’s a programming block?

A: One or more statements surrounded by curly braces that form a single unit.

Q: What’s a compiler warning?

A: A message from your compiler stating a potential problem. A warning will not stop the compilation process.

Q: Can I ignore compiler warnings?

A: You can, but you shouldn’t. You should address the warning and fix the offending code.

Q: What is whitespace?

A: A set of non-printing characters that create space in your source files, including tabs, spaces, and newlines.

Q: What are literals?

A: Elements that represent explicit values. "Game Over!" is a string literal, while

32 and 98.6 are numeric literals.

Q: Why should I always try to initialize a new variable with a value?

A: Because the contents of an uninitialized variable could be any value—even one that doesn’t make sense for your program.

Q: Why do programmers sometimes use variable names such as myInt or

myFloat?

A: To clearly spell out a variable’s type. This convention is used frequently in programming instruction.

Q: What are variables of type bool for?

A: They can represent a condition that is true or false, such as whether a chest is locked or a playing card is face up.

38 Chapter 1 n Types, Variables, and Standard I/O: Lost Fortune

Q: How did the bool type get its name?

A: The type is named in honor of the English mathematician George Boole. Q: Must the names of constants be in uppercase letters?

A: No. Using uppercase is just an accepted practice—but one you should use because it’s what other programmers expect.

Q: How can I store more than one character with a single variable? A: With a string object.

### [Discussion Questions](#_bookmark2)

1. How does having a widely adopted Cþþ standard help game programmers?
2. What are the advantages and disadvantages of employing the using

directive?

1. Why might you define a new name for an existing type?
2. Why are there two versions of the increment operator? What’s the difference between them?
3. How can you use constants to improve your code?

### [Exercises](#_bookmark2)

1. Create a list of six legal variable names—three good and three bad choices. Explain why each name falls into the good or bad category.
2. What’s displayed by each line in the following code snippet? Explain each result.

cout << "Seven divided by three is " << 7 / 3 << endl; cout << "Seven divided by three is " << 7.0 / 3 << endl; cout << "Seven divided by three is " << 7.0 / 3.0 << endl;

1. Write a program that gets three game scores from the user and displays the average.

|  |  |
| --- | --- |
|  |  |
| [chapter 2](#_bookmark2) | |

# [Truth, Branching, and the Game Loop: Guess My Number](#_bookmark2)

So far, the programs you’ve seen have been linear—each statement executes, in order, from top to bottom. However, to create interesting games, you need to write programs that execute (or skip) sections of code based on some condition. That’s the main topic of this chapter. Specifically, you’ll learn to:

n Understand truth (as C++ defines it)

n Use if statements to branch to sections of code

n Use switch statements to select a section of code to execute

n Use while and do loops to repeat sections of code

n Generate random numbers

### [Understanding Truth](#_bookmark2)

Truth is black and white, at least as far as C++ is concerned. You can represent true and false with their corresponding keywords, true and false. You can store such a Boolean value with a bool variable, as you saw in Chapter 1. Here’s a quick refresher:

bool fact = true, fiction = false;

This code creates two bool variables, fact and fiction. fact is true and fiction

is false. Although the keywords true and false are handy, any expression or

39

40 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

value can be interpreted as true or false, too. Any non-zero value can be interpreted as true, while 0 can be interpreted as false.

A common kind of expression interpreted as true or false involves comparing things. Comparisons are often made by using built-in relational operators. Table 2.1 lists the operators and a few sample expressions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Table 2.1** Relational Operators | |  | |  |
| **Operator** | **Meaning** | **Sample Expression** | | **Evaluates To** |
| == | equal to | 5 | == 5 | true |
|  | | 5 | == 8 | false |
| != | not equal to | 5 | != 8 | true |
|  | | 5 | != 5 | false |
| > greater than | | 8 | > 5 | true |
|  | | 5 | > 8 | false |
| < | less than | 5 | < 8 | true |
|  | | 8 | < 5 | false |
| >= | greater than or equal to | 8 | >= 5 | true |
|  | | 5 | >= 8 | false |
| <= | less than or equal to | 5 | <= 8 | true |
|  | | 8 | <= 5 | false |

### [Using the if Statement](#_bookmark3)

Okay, it’s time to put the concepts of true and false to work. You can use an if statement to test an expression for truth and execute some code based on it. Here’s a simple form of the if statement:

if (*expression*)

*statement*;

If *expression* is true, then *statement* is executed. Otherwise, *statement* is skipped and the program branches to the statement after the if suite.

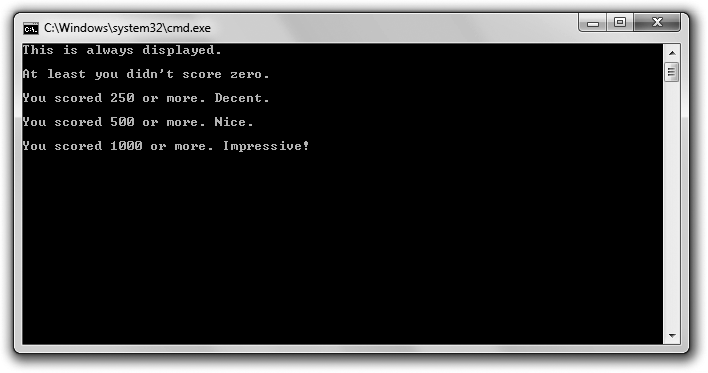
**Hi n t**

Whenever you see a generic *statement* like in the preceding code example, you can replace it with a single statement or a block of statements because a block is treated as a single unit.

Using the if Statement 41

#### [Introducing the Score Rater Program](#_bookmark3)

The Score Rater program comments on a player’s score using an if statement. Figure 2.1 shows the program in action.



**Figure 2.1**

Messages are displayed (or not displayed) based on different if statements.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 2 folder; the filename is score\_rater.cpp.

// Score Rater

// Demonstrates the if statement

#include <iostream> using namespace std;

int main()

{

if (true)

{

cout << "This is always displayed.\n\n";

}

42 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

if (false)

{

cout << "This is never displayed.\n\n";

}

int score = 1000;

if (score)

{

cout << "At least you didn’t score zero.\n\n";

}

if (score >= 250)

{

cout << "You scored 250 or more. Decent.\n\n";

}

if (score >= 500)

{

cout << "You scored 500 or more. Nice.\n\n";

if (score >= 1000)

{

cout << "You scored 1000 or more. Impressive!\n";

}

}

return 0;

}

#### [Testing true and false](#_bookmark3)

In the first if statement I test true. Because true is, well, true, the program displays the message, “This is always displayed.”

if (true)

{

cout << "This is always displayed.\n\n";

}

In the next if statement I test false. Because false isn’t true, the program doesn’t display the message, “This is never displayed.”

Using the if Statement 43

if (false)

{

cout << "This is never displayed.\n\n";

}

**Trap**

Notice that you don’t use a semicolon after the closing parenthesis of the expression you test in an if statement. If you were to do this, you’d create an empty statement that would be paired with the if statement, essentially rendering the if statement useless. Here’s an example:

if (false);

{

cout << "This is never displayed.\n\n";

}

By adding the semicolon after (false), I create an empty statement that’s associated with the

if statement. The preceding code is equivalent to:

if (false)

; // an empty statement, which does nothing

{

cout << "This is never displayed.\n\n";

}

All I’ve done is play with the whitespace, which doesn’t change the meaning of the code. Now the problem should be clear. The if statement sees the false value and skips the next statement (the empty statement). Then the program goes on its merry way to the statement after the if statement, which displays the message, “This is never displayed.”

Be on guard for this error. It’s an easy one to make and because it’s not illegal, it won’t produce a compile error.

#### [Interpreting a Value as true or false](#_bookmark3)

You can interpret any value as true or false. Any non-zero value can be interpreted as true, while 0 can be interpreted as false. I put this to the test in the next if statement:

if (score)

{

cout << "At least you didn’t score zero.\n\n";

}

score is 1000, so it’s non-zero and interpreted as true. As a result, the message,

“Okay, at least you didn’t score zero,” is displayed.

44 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

#### [Using Relational Operators](#_bookmark3)

Probably the most common expression you’ll use with if statements involves comparing values using the relational operators. That’s just what I’ll demon- strate next. I test to see whether the score is greater than or equal to 250.

if (score >= 250)

{

cout << "You scored 250 or more. Decent.\n\n";

}

Because score is 1000, the block is executed, displaying the message that the player earned a decent score. If score had been less than 1000, the block would have been skipped and the program would have continued with the statement following the block.

**Trap**

The equal to relational operator is == (two equal signs in a row). Don’t confuse it with = (one equal sign), which is the assignment operator.

While it’s not illegal to use the assignment operator instead of the equal to relational operator, the results might not be what you expect. Take a look at this code:

int score = 500; if (score = 1000)

{

cout << " You scored 1000 or more. Impressive!\n";

}

As a result of this code, score is set to 1000 and the message, “You scored 1000 or more. Impressive!” is displayed. Here’s what happens: Although score is 500 before the if statement, that changes. When the expression of the if statement, (score = 1000), is evaluated, score is assigned 1000. The assignment statement evaluates to 1000, and because that’s a non-zero value, the expression is interpreted as true. As a result, the string is displayed.

Be on guard for this type of mistake. It’s easy to make and in some cases (like this one), it won’t cause a compile error.

#### [Nesting if Statements](#_bookmark3)

An if statement can cause a program to execute a statement or block of statements, including other if statements. When you write one if statement

Using the else Clause 45

inside another, it’s called *nesting*. In the following code, the if statement that begins if (score >= 1000) is nested inside the if statement that begins if (score > 500).

if (score >= 500)

{

cout << "You scored 500 or more. Nice.\n\n";

if (score >= 1000)

{

cout << "You scored 1000 or more. Impressive!\n";

}

}

Because score is greater than 500, the program enters the statement block and displays the message, “You scored 500 or more. Nice.” Then, in the inner if statement, the program compares score to 1000. Because score is greater than or equal to 1000, the program displays the message, “You scored 1000 or more. Impressive!”

**Hint**

You can nest as many levels as you want. However, if you nest code too deeply, it gets hard to read. In general, you should try to limit your nesting to a few levels at most.

### [Using the else Clause](#_bookmark3)

You can add an else clause to an if statement to provide code that will only be executed if the tested expression is false. Here’s the form of an if statement that includes an else clause:

if (*expression*)

*statement1*; else

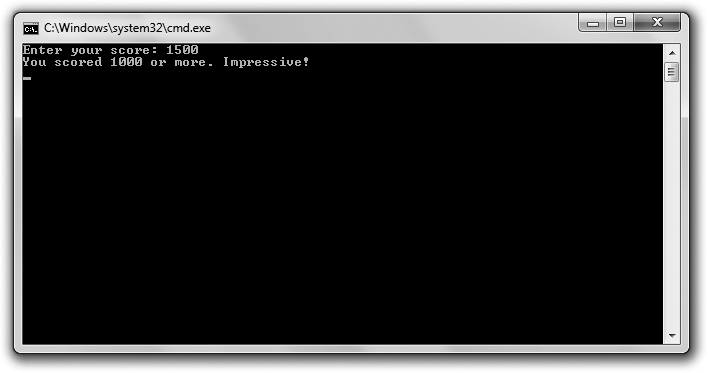
*statement2*;

If *expression* is true, *statement1* is executed. Then the program skips *statement2* and executes the statement following the if suite. If *expression* is false, *statement1* is skipped and *statement2* is executed. After *statement2* completes, the program executes the statement following the if suite.

46 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

#### [Introducing the Score Rater 2.0 Program](#_bookmark3)

The Score Rater 2.0 program also rates a score, which the user enters. But this time, the program uses an if statement with an else clause. Figures 2.2 and 2.3 show the two different messages that the program can display based on the score the user enters.



**Figure 2.2**

If the user enters a score that’s 1000 or more, he is congratulated.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 2 folder; the filename is score\_rater2.cpp.

// Score Rater 2.0

// Demonstrates an else clause

#include <iostream> using namespace std;

int main()

{

int score;

cout << "Enter your score: ";

Using the else Clause 47

cin >> score;

if (score >= 1000)

{

}

else

{

}

cout << "You scored 1000 or more. Impressive!\n";

cout << "You scored less than 1000.\n";

return 0;

}



**Figure 2.3**

If the user enters a score that’s less than 1000, there’s no celebration.

#### [Creating Two Ways to Branch](#_bookmark3)

You’ve seen the first part of the if statement already, and it works just as it did before. If score is greater than 1000, the message, “You scored 1000 or more. Impressive!” is displayed.

48 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

if (score >= 1000)

{

cout << "You scored 1000 or more. Impressive!\n";

}

Here’s the twist. The else clause provides a statement for the program to branch to if the expression is false. So if (score >= 1000) is false, then the program skips the first message and instead displays the message, “You scored less than 1000.”

else

{

cout << "You scored less than 1000.\n";

}

### Using a Sequence of if Statements [with else Clauses](#_bookmark3)

You can chain if statements with else clauses together to create a sequence of expressions that get tested in order. The statement associated with the first expression to test true is executed; otherwise, the statement associated with the final (and optional) else clause is run. Here’s the form such a series would take:

if (*expression1*)

*statement1*;

else if (expression2)

*statement2*;

...

else if (*expressionN*)

*statementN*;

else

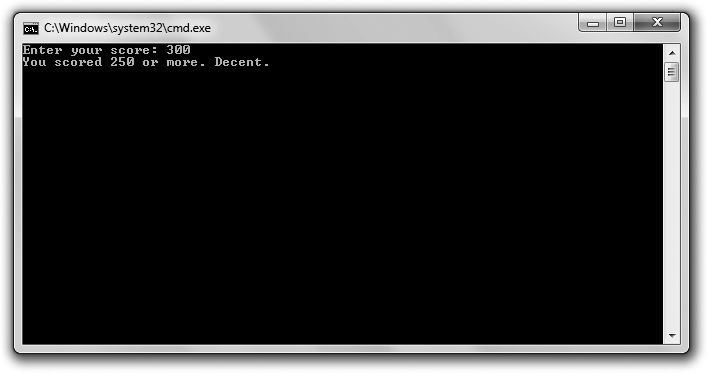
statementN+1;

If *expression1* is true, *statement1* is executed and the rest of the code in the sequence is skipped. Otherwise, *expression2* is tested and if true, *statement2* is executed and the rest of the code in the sequence is skipped. The computer continues to check each expression in order (through *expressionN*) and will execute the statement associated with the first expression that is true. If no expression is true, then the statement associated with the final else clause, *statementN+1*, is executed.

Using a Sequence of if Statements with else Clauses 49

#### [Introducing the Score Rater 3.0 Program](#_bookmark3)

The Score Rater 3.0 program also rates a score, which the user enters. But this time, the program uses a sequence of if statements with else clauses. Figure 2.4 shows the results of the program.



**Figure 2.4**

The user can get one of multiple messages, depending on his score.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 2 folder; the filename is score\_rater3.cpp.

// Score Rater 3.0

// Demonstrates if else-if else suite

#include <iostream> using namespace std;

int main()

{

int score;

cout << "Enter your score: "; cin >> score;

50 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

if (score >= 1000)

{

cout << "You scored 1000 or more. Impressive!\n";

}

else if (score >= 500)

{

cout << "You scored 500 or more. Nice.\n";

}

else if (score >= 250)

{

cout << "You scored 250 or more. Decent.\n";

}

else

{

cout << "You scored less than 250. Nothing to brag about.\n";

}

return 0;

}

[**Creating a Sequence of if Statements with else Clauses**](#_bookmark3)You’ve seen the first part of this sequence twice already, and it works just the same this time around. If score is greater than or equal to 1000, the message,

“You scored 1000 or more. Impressive!” is displayed and the computer branches

to the return statement.

if (score >= 1000)

However, if the expression is false, then we know that score is less than 1000 and the computer evaluates the next expression in the sequence:

else if (score >= 500)

If score is greater than or equal to 500, the message, “You scored 500 or more. Nice.” is displayed and the computer branches to the return statement. However, if that expression is false, then we know that score is less than 500 and the computer evaluates the next expression in the sequence:

else if (score >= 250)

If score is greater than or equal to 250, the message, “You scored 250 or more. Decent.” is displayed and the computer branches to the return statement.

Using the switch Statement 51

However, if that expression is false, then we know that score is less than 250 and the statement associated with the final else clause is executed and the message, “You scored less than 250. Nothing to brag about.” is displayed.

**Hint**

While the final else clause in an if else-if suite isn’t required, you can use it as a way to execute code if none of the expressions in the sequence are true.

### [Using the switch Statement](#_bookmark3)

You can use a switch statement to create multiple branching points in your code. Here’s a generic form of the switch statement:

switch (*choice*)

{

case *value1*: *statement1*;

break;

case *value2*: *statement2*;

break;

case *value3*: *statement3*;

break;

.

.

.

case *valueN*: *statementN;*

break;

default: *statementN + 1*;

}

The statement tests *choice* against the possible values—*value1*, *value2*, and *value3*—in order. If *choice* is equal to a value, then the program executes the corresponding *statement*. When the program hits a break statement, it exits the switch structure. If *choice* doesn’t match any value, then the statement associated with the optional default is executed.

The use of break and default are optional. If you leave out a break, however, the program will continue through the remaining statements until it hits a break or a default or until the switch statement ends. Usually you want one break statement to end each case.

52 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

**Hi n t**

Although a default case isn’t required, it’s usually a good idea to have one as a catchall.

Here’s an example to cement the ideas. Suppose *choice* is equal to *value2*. The program will first test *choice* against *value1*. Because they’re not equal, the program will continue. Next, the program will test *choice* against *value2*. Because they are equal, the program will execute *statement2*. Then the program will hit the break statement and exit the switch structure.

**Trap**

You can use the switch statement only to test an int (or a value that can be treated as an int, such as a char or an enumerator). A switch statement won’t work with any other type.

#### [Introducing the Menu Chooser Program](#_bookmark3)

The Menu Chooser program presents the user with a menu that lists three difficulty levels and asks him to make a choice. If the user enters a number that corresponds to a listed choice, then he is shown a message confirming the choice. If the user makes some other choice, he is told that the choice is invalid. Figure 2.5 shows the program in action.



**Figure 2.5**

Looks like I took the easy way out.

Using the switch Statement 53

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 2 folder; the filename is menu\_chooser.cpp.

// Menu Chooser

// Demonstrates the switch statement #include <iostream>

using namespace std;

int main()

{

cout << "Difficulty Levels\n\n"; cout << "1 - Easy\n";

cout << "2 - Normal\n"; cout << "3 - Hard\n\n";

int choice;

cout << "Choice: "; cin >> choice;

switch (choice)

{

case 1:

case 2:

case 3:

cout << "You picked Easy.\n"; break;

cout << "You picked Normal.\n"; break;

cout << "You picked Hard.\n"; break;

default:

cout << "You made an illegal choice.\n";

}

return 0;

}

54 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

#### [Creating Multiple Ways to Branch](#_bookmark3)

The switch statement creates four possible branching points. If the user enters 1, then code associated with case 1 is executed and “You picked Easy” is displayed. If the user enters 2, then code associated with case 2 is executed and “You picked Normal” is displayed. If the user enters 3, then code associated with case 3 is executed and “You picked Hard” is displayed. If the user enters any other value, then default kicks in and “You made an illegal choice” is displayed.

**Trap**

You’ll almost always want to end each case with a break statement. Don’t forget them; otherwise, your code might do things you never intended.

### [Using while Loops](#_bookmark3)

while loops let you repeat sections of code as long as an expression is true. Here’s a generic form of the while loop:

while (*expression*)

*statement*;

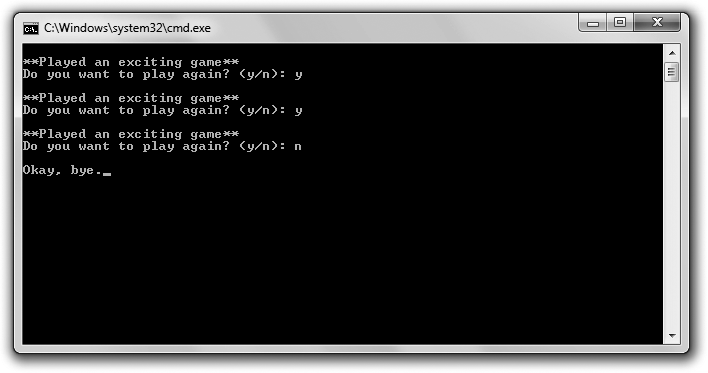
If *expression* is false, the program moves on to the statement after the loop. If *expression* is true, the program executes *statement* and loops back to test *expression* again. This cycle repeats until *expression* tests false, at which point the loop ends.

#### [Introducing the Play Again Program](#_bookmark3)

The Play Again program simulates the play of an exciting game. (Okay, by “simulates the play of an exciting game,” I mean the program displays the message “\*\*Played an exciting game\*\*”.) Then the program asks the user if he wants to play again. The user continues to play as long as he enters y. The program accomplishes this repetition using a while loop. Figure 2.6 shows the program in action.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 2 folder; the filename is play\_again.cpp.

Using while Loops 55



**Figure 2.6**

The repetition is accomplished using a while loop.

// Play Again

// Demonstrates while loops

#include <iostream> using namespace std;

int main()

{

char again = ’y’; while (again == ’y’)

{

cout << "\n\*\*Played an exciting game\*\*";

cout << "\nDo you want to play again? (y/n): "; cin >> again;

}

cout << "\nOkay, bye.";

return 0;

}

#### [Looping with a while Loop](#_bookmark3)

The first thing the program does in the main() function is declare the char

variable named again and initialize it to ’y’. Then the program begins the while

56 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

loop by testing again to see whether it’s equal to ’y’. Because it is, the program displays the message “\*\*Played an exciting game\*\*,” asks the user whether he wants to play again, and stores the reply in again. The loop continues as long as the user enters y.

You’ll notice that I had to initialize again before the loop because the variable is used in the loop expression. Because a while loop evaluates its expressions before its *loop body* (the group of statements that repeat), you have to make sure that any variables in the expression have a value before the loop begins.

### [Using do Loops](#_bookmark3)

Like while loops, do loops let you repeat a section of code based on an expression. The difference is that a do loop tests its expression after each loop iteration. This means that the loop body is always executed at least once. Here’s a generic form of a do loop:

do

*statement*; while (*expression*);

The program executes *statement* and then, as long as *expression* tests true, the loop repeats. Once *expression* tests false, the loop ends.

#### [Introducing the Play Again 2.0 Program](#_bookmark3)

The Play Again 2.0 program looks exactly the same to the user as the original Play Again program. Play Again 2.0, like its predecessor, simulates the play of an exciting game by displaying the message “\*\*Played an exciting game\*\*” and asking the user whether he wants to play again. The user continues to play as long as he enters y. This time, though, the program accomplishes the repetition using a do loop. Figure 2.7 shows off the program.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 2 folder; the filename is play\_again2.cpp.

// Play Again 2.0

// Demonstrates do loops #include <iostream>

Using do Loops 57

using namespace std; int main()

{

char again; do

{

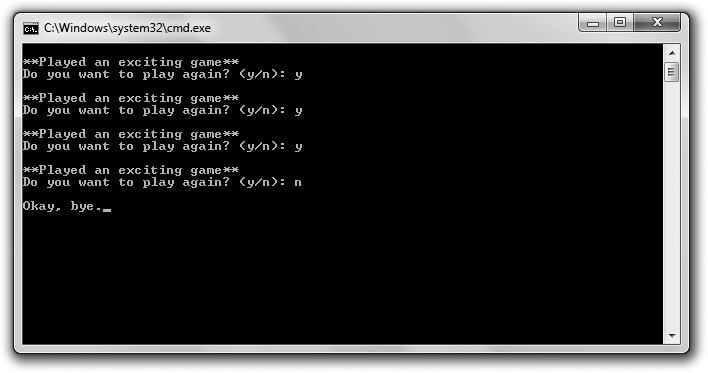
cout << "\n\*\*Played an exciting game\*\*";

cout << "\nDo you want to play again? (y/n): "; cin >> again;

} while (again == ’y’); cout << "\nOkay, bye.";

return 0;

}



**Figure 2.7**

Each repetition is accomplished using a do loop.

#### [Looping with a do Loop](#_bookmark3)

Before the do loop begins, I declare the character again. However, I don’t need to initialize it because it’s not tested until after the first iteration of the loop. I get a

58 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

new value for again from the user in the loop body. Then I test again in the loop expression. If again is equal to ’y’, the loop repeats; otherwise, the loop ends.

**IRW**

Even though you can use while and do loops pretty interchangeably, most programmers use the while loop. Although a do loop might seem more natural in some cases, the advantage of a while loop is that its expression appears right at the top of the loop; you don’t have to go hunting to the bottom of the loop to find it.

**Trap**

If you’ve ever had a game get stuck in the same endless cycle, you might have experienced an

*infinite loop*—a loop without end. Here’s a simple example of an infinite loop:

int test = 10; while (test == 10)

{

cout << test;

}

In this case, the loop is entered because test is 10. But because test never changes, the loop will never stop. As a result, the user will have to kill the running program to end it. The moral of this story? Make sure that the expression of a loop can eventually become false or that there’s another way for the loop to end, such as described in the following section, “Using break and continue Statements.”

### [Using break and continue Statements](#_bookmark3)

It’s possible to alter the behavior you’ve seen in loops. You can immediately exit a loop with the break statement, and you can jump directly to the top of a loop with a continue statement. Although you should use these powers sparingly, they do come in handy sometimes.

#### [Introducing the Finicky Counter Program](#_bookmark3)

The Finicky Counter program counts from 1 to 10 through a while loop. It’s finicky because it doesn’t like the number 5—it skips it. Figure 2.8 shows a run of the program.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 2 folder; the filename is finicky\_counter.cpp.

Using break and continue Statements 59



**Figure 2.8**

The number 5 is skipped with a continue statement, and the loop ends with a break statement.

// Finicky Counter

// Demonstrates break and continue statements #include <iostream>

using namespace std;

int main()

{

int count = 0; while (true)

{

count += 1;

//end loop if count is greater than 10 if (count > 10)

{

break;

}

60 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

//skip the number 5 if (count == 5)

{

continue;

}

cout << count << endl;

}

return 0;

}

#### [Creating a while (true) Loop](#_bookmark3)

I set up the loop with the following line:

while (true)

Technically, this creates an infinite loop. This might seem odd coming so soon after a warning to avoid infinite loops, but this particular loop isn’t really infinite because I put an exit condition in the loop body.

**Hi n t**

Although a while (true) loop sometimes can be clearer than a traditional loop, you should also try to minimize your use of these loops.

#### [Using the break Statement to Exit a Loop](#_bookmark3)

This is the exit condition I put in the loop:

//end loop if count is greater than 10 if (count > 10)

{

break;

}

Because count is increased by 1 each time the loop body begins, it will eventually reach 11. When it does, the break statement (which means “break out of the loop”) is executed and the loop ends.

Using Logical Operators 61

#### [Using the continue Statement to Jump Back to the Top of a Loop](#_bookmark3)

Just before count is displayed, I included the lines:

//skip the number 5 if (count == 5)

{

continue;

}

The continue statement means “jump back to the top of the loop.” At the top of the loop, the while expression is tested and the loop is entered again if it’s true. So when count is equal to 5, the program does not get to the cout << count << endl; statement. Instead, it goes right back to the top of the loop. As a result, 5 is skipped and never displayed.

#### [Understanding When to Use break and continue](#_bookmark3)

You can use break and continue in any loop you create; they aren’t just for while (true) loops. But you should use them sparingly. Both break and continue can make it harder for programmers to see the flow of a loop.

### [Using Logical Operators](#_bookmark3)

So far you’ve seen pretty simple expressions evaluated for their truth or falsity. However, you can combine simpler expressions with *logical operators* to create more complex expressions. Table 2.2 lists the logical operators.

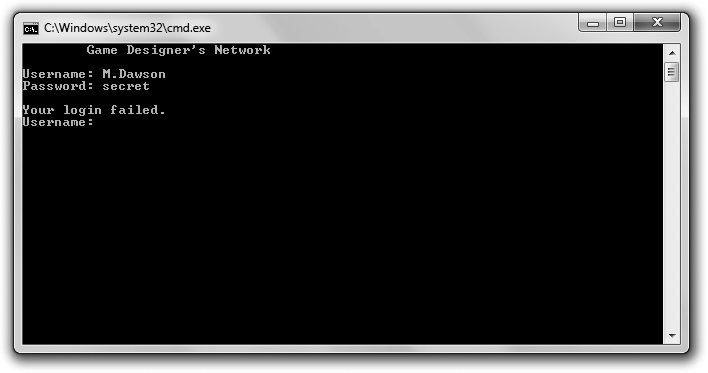
|  |  |  |
| --- | --- | --- |
| **Table 2.2** | Logical Operators |  |
| **Operator** | **Description** | **Sample Expression** |
| ! | Logical NOT | !*expression* |
| && | Logical AND | *expression1* && *expression2* |
| || | Logical OR | *expression1* || *expression2* |

#### [Introducing the Designers Network Program](#_bookmark3)

The Designers Network program simulates a computer network in which only a select group of game designers are members. Like real-world computer systems,

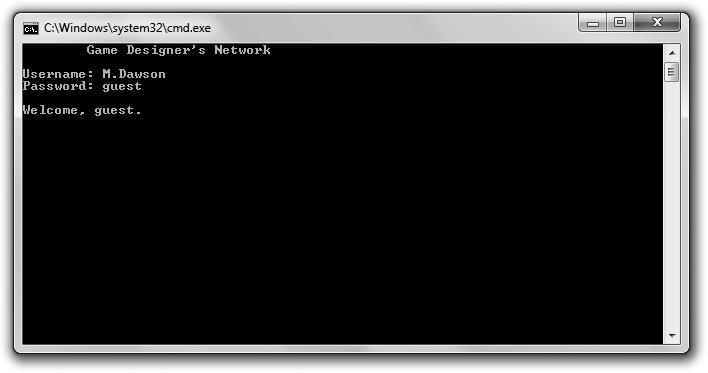
62 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

each member must enter a username and a password to log in. With a successful login, the member is personally greeted. To log in as a guest, all a user needs to do is enter guest at either the username or password prompt. Figures 2.9 through 2.11 show the program.



**Figure 2.9**

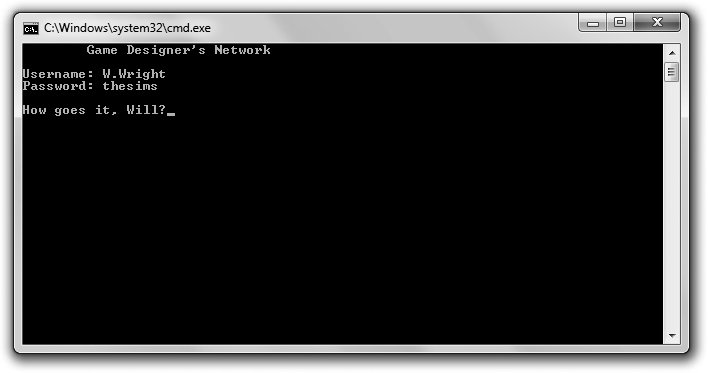
If you’re not a member or a guest, you can’t get in.



**Figure 2.10**

You can log in as a guest.

Using Logical Operators 63



**Figure 2.11**

Looks like one of the elite logged in today.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 2 folder; the filename is designers\_network.cpp.

// Designers Network

// Demonstrates logical operators

#include <iostream> #include <string> using namespace std;

int main()

{

string username; string password; bool success;

cout << "\tGame Designer’s Network\n"; do

64 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

{

cout << "\nUsername: "; cin >> username;

cout << "Password: "; cin >> password;

if (username == "S.Meier" && password == "civilization")

{

cout << "\nHey, Sid."; success = true;

}

else if (username == "S.Miyamoto" && password == "mariobros")

{

cout << "\nWhat’s up, Shigeru?"; success = true;

}

else if (username == "W.Wright" && password == "thesims")

{

cout << "\nHow goes it, Will?"; success = true;

}

else if (username == "guest" || password == "guest")

{

}

else

{

}

cout << "\nWelcome, guest."; success = true;

cout << "\nYour login failed."; success = false;

} while (!success);

return 0;

}

Using Logical Operators 65

#### [Using the Logical AND Operator](#_bookmark3)

The logical AND operator, &&, lets you join two expressions to form a larger one, which can be evaluated to true or false. The new expression is true only if the two expressions it joins are true; otherwise, it is false. Just as in English, “and” means both. Both original expressions must be true for the new expression to be true. Here’s a concrete example from the Designers Network program:

if (username == "S.Meier" && password == "civilization")

The expression username == "S.Meier" && password == "civilization" is true only if both username == "S.Meier" and password == "civilization" are true. This works perfectly because I only want to grant Sid access if he enters both his username and his password. Just one or the other won’t do.

Another way to understand how && works is to look at all of the possible combinations of truth and falsity (see Table 2.3).

true

false

false false

true

false

true false

true

true

false false

**username == "S.Meier" &&**

**password == "civilization"**

**password ==**

**"civilization"**

**username ==**

**"S.Meier"**

**Table 2.3** Possible Login Combinations Using the AND Operator

Of course, the Designers Network program works for other users besides Sid Meier. Through a series of if statements with else clauses using the && operator, the program checks three different username and password pairs. If a user enters a recognized pair, he is personally greeted.

#### [Using the Logical OR Operator](#_bookmark3)

The logical OR operator, ||, lets you join two expressions to form a larger one, which can be evaluated to true or false. The new expression is true if the first

66 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

expression *or* the second expression is true; otherwise, it is false. Just as in English, “or” means either. If either the first or second expression is true, then the new expression is true. (If both are true, then the larger expression is still true.) Here’s a concrete example from the Designers Network program:

else if (username == "guest" || password == "guest")

The expression username == "guest" || password == "guest" is true if username == "guest" is true or if password == "guest" is true. This works perfectly because I want to grant a user access as a guest as long as he enters guest for the username or password. If the user enters guest for both, that’s fine too.

Another way to understand how || works is to look at all of the possible combinations of truth and falsity (see Table 2.4).

true

true

true false

true

false

true false

true

true

false false

**username == "guest" ||**

**password == "guest"**

**password ==**

**"guest"**

**username ==**

**"guest"**

**Table 2.4** Possible Login Combinations Using the OR Operator

#### [Using the Logical NOT Operator](#_bookmark3)

The logical NOT operator, !, lets you switch the truth or falsity of an expression. The new expression is true if the original is false; the new expression is false if the original is true. Just as in English, “not” means the opposite. The new expression has the opposite value of the original.

I use the NOT operator in the Boolean expression of the do loop:

} while (!success);

The expression !success is true when success is false. That works perfectly because success is false only when there has been a failed login. In that case,

Using Logical Operators 67

the block associated with the do loop executes again and the user is asked for his username and password once more.

The expression !success is false when success is true. That works perfectly because when success is true, the user has successfully logged in and the loop ends.

Another way to understand how ! works is to look at all of the possible combinations of truth and falsity (see Table 2.5).

false

true

true

false

**!security**

**security**

**Table 2.5** Possible Login Combinations Using

the NOT Operator

#### [Understanding Order of Operations](#_bookmark3)

Just like arithmetic operators, logical operators have precedence levels that affect the order in which an expression is evaluated. Logical NOT, !, has a higher level of precedence than logical AND, &&, which has a higher precedence than logical OR, ||.

Just as with arithmetic operators, if you want an operation with lower precedence to be evaluated first, you can use parentheses. You can create complex expressions that involve arithmetic operators, relational operators, and logical operators. Operator precedence will define the exact order in which elements of the expression are evaluated. However, it’s best to try to create expressions that are clear and simple, not ones that require a mastery of the operator precedence list to decipher.

For a list of C++ operators and their precedence levels, see Appendix B.

68 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

**Hi n t**

Although you can use parentheses in a larger expression to change the way in which it’s evaluated, you can also use *redundant parentheses*—parentheses that don’t change the value of the expressions—to make the expression clearer. Let me give you a simple example. Check out the following expression from the Designers Network program:

(username == "S.Meier" && password == "civilization")

Now, here’s the expression with some redundant parentheses:

( (username == "S.Meier") && (password == "civilization") )

While the extra parentheses don’t change the meaning of the expression, they really help the two smaller expressions, joined by the && operator, stand out.

Using redundant parentheses is a bit of an art form. Are they helpful or just plain redundant? That’s a call you as the programmer have to make.

### [Generating Random Numbers](#_bookmark3)

A sense of unpredictability can add excitement to a game. Whether it’s the sudden change in a computer opponent’s strategy in an RTS or an alien creature bursting from an arbitrary door in an FPS, players thrive on a certain level of surprise. Generating random numbers is one way to achieve this kind of surprise.

#### [Introducing the Die Roller Program](#_bookmark3)

The Die Roller program simulates the roll of a six-sided die. The computer calculates the roll by generating a random number. Figure 2.12 shows the results of the program.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 2 folder; the filename is die\_roller.cpp.

// Die Roller

// Demonstrates generating random numbers

#include <iostream> #include <cstdlib> #include <ctime>

using namespace std;

Generating Random Numbers 69



**Figure 2.12**

The die roll is based on a random number generated by the program.

int main()

{

srand(static\_cast<unsigned int>(time(0))); //seed random number generator int randomNumber = rand(); //generate random number

int die = (randomNumber % 6) + 1; // get a number between 1 and 6 cout << "You rolled a " << die << endl;

return 0;

}

#### [Calling the rand() Function](#_bookmark3)

One of the first things I do in the program is include a new file:

#include <cstdlib>

The file cstdlib contains (among other things) functions that deal with generating random numbers. Because I’ve included the file, I’m free to call

70 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

the functions it contains, including the function rand(), which is exactly what I do in main():

int randomNumber = rand(); //generate random number

As you learned in Chapter 1, functions are pieces of code that can do some work and return a value. You call or invoke a function by using its name followed by a pair of parentheses. If a function returns a value, you can assign that value to a variable. That’s what I do here with my use of the assignment statement. I assign the value returned by rand() (a random number) to randomNumber.

**Hi n t**

The rand() function generates a random number between 0 and at least 32767. The exact upper limit depends on your implementation of C++. The upper limit is stored in the constant RAND\_MAX, which is defined in cstdlib. So if you want to know the maximum random number rand() can generate, just send RAND\_MAX to cout.

Functions can also take values to use in their work. You provide these values by placing them between the parentheses after the function name, separated by commas. These values are called *arguments*, and when you provide them, you *pass* them to the function. I didn’t pass any values to rand() because the function doesn’t take any arguments.

#### [Seeding the Random Number Generator](#_bookmark3)

Computers generate *pseudorandom* numbers—not truly random numbers— based on a formula. One way to think about this is to imagine that the computer reads from a huge book of predetermined numbers. By reading from this book, the computer can appear to produce a sequence of random numbers.

But there’s a problem: The computer always starts reading the book from the beginning. Because of this, the computer will always produce the same series of “random” numbers in a program. In games, this isn’t something we’d want. We wouldn’t, for example, want the same series of dice rolls in a game of craps every time we played.

A solution to this problem is to tell the computer to start reading from some arbitrary place in the book when a game program begins. This process is called

Generating Random Numbers 71

*seeding* the random number generator. Game programmers give the random number generator a number, called a *seed*, to determine the starting place in this sequence of pseudorandom numbers.

The following code seeds the random number generator:

srand(static\_cast<unsigned int>(time(0))); //seed random number generator

Wow, that’s a pretty cryptic looking line, but what it does is simple. It seeds the random number generator based on the current date and time, which is perfect since the current date and time will be different for each run of the program.

In terms of the actual code, the srand() function seeds the random number generator—you just have to pass it an unsigned int as a seed. What gets passed to the function here is the return value of time(0)—a number based on the current system date and time. The code static\_cast<unsigned int> just converts (or *casts*) this value to an unsigned int. Now, you don’t have to understand all the nuances of this line; the least you need to know is that if you want a program to generate a series of random numbers that are different each time the program is run, your program should execute this line once before calls to rand().

**Hint**

A comprehensive explanation of the various forms of casting a value from one type to another is beyond the scope of this book.

#### [Calculating a Number within a Range](#_bookmark3)

After generating a random number, randomNumber holds a value between 0 and 32767 (based on my implementation of C++). But I need a number between 1 and 6, so next I use the modulus operator to produce a number in that range.

int die = (randomNumber % 6) + 1; // get a number between 1 and 6

Any positive number divided by 6 will give a remainder between 0 and 5. In the preceding code, I take this remainder and add 1, giving me the possible range of 1 through 6—exactly what I wanted. You can use this technique to convert a random number to a number within a range you’re looking for.

72 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

**Trap**

Using the modulus operator to create a number within a range from a random number might not always produce uniform results. Some numbers in the range might be more likely to appear than others. However, this isn’t a problem for simple games.

### [Understanding the Game Loop](#_bookmark4)

The *game loop* is a generalized representation of the flow of events in a game. The core of the events repeats, which is why it’s called a loop. Although the implementation might be quite different from game to game, the fundamental structure is the same for almost all games across genres. Whether you’re talking about a simple space shooter or a complex role-playing game (RPG), you can usually break the game down into the same repeating components of the game loop. Figure 2.13 provides a visual representation of the game loop.

Here’s an explanation of the parts of the game loop:

n **Setup.** This often involves accepting initial settings or loading game assets, such as sound, music, and graphics. The player might also be pre- sented with the game backstory and his objectives.

n **Getting player input.** Whether it comes from the keyboard, mouse, joy- stick, trackball, or some other device, input from the player is captured.

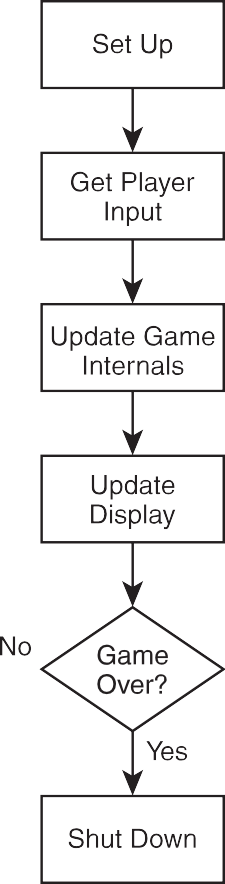
n **Updating game internals.** The game logic and rules are applied to the game world, taking into account player input. This might take the shape of a physics system determining the interaction of objects or it might involve calculations of enemy AI, for example.

n **Updating the display.** In the majority of games, this process is the most taxing on the computer hardware because it often involves drawing graphics. However, this process can be as simple as displaying a line of text.

n **Checking whether the game is over.** If the game isn’t over (if the play- er’s character is still alive and the player hasn’t quit, for example), con- trol branches back to the getting player input stage. If the game is over, control falls through to the shutting down stage.

n **Shutting down.** At this point, the game is over. The player is often given some final information, such as his score. The program frees any resources, if necessary, and exits.

Introducing Guess My Number 73



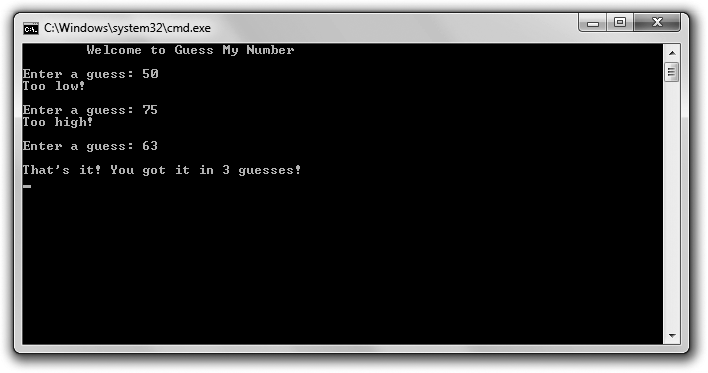
**Figure 2.13**

The game loop describes a basic flow of events that fits just about any game.

### [Introducing Guess My Number](#_bookmark4)

The final project for this chapter, Guess My Number, is the classic number- guessing game. For those who missed out on this game in their childhood, it goes like this: The computer chooses a random number between 1 and 100, and the player tries to guess the number in as few attempts as possible. Each time the player enters a guess, the computer tells him whether the guess is too high, too low, or right on the money. Once the player guesses the number, the game is over. Figure 2.14 shows Guess My Number in action. You can download the code for this program from the Course Technology website [(www.courseptr.](http://www.courseptr.com/downloads) [com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 2 folder; the filename is guess\_my\_number.cpp.

74 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number



**Figure 2.14**

I guessed the computer’s number in just three tries.

#### [Applying the Game Loop](#_bookmark4)

It’s possible to examine even this simple game through the construct of the game loop. Figure 2.15 shows how nicely the game loop paradigm fits the flow of the game.

#### [Setting Up the Game](#_bookmark4)

As always, I start off with some comments and include the necessary files.

// Guess My Number

// The classic number guessing game

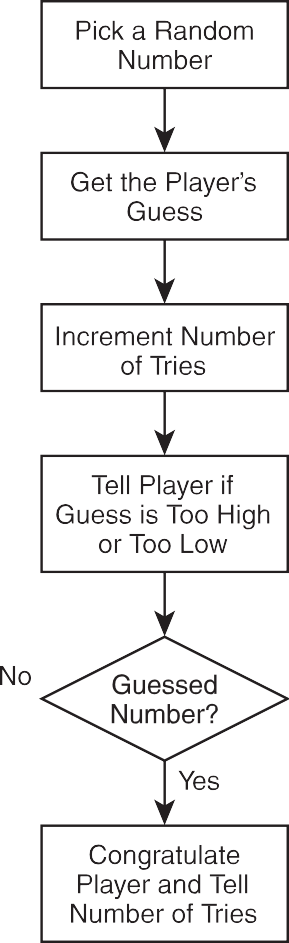
#include <iostream> #include <cstdlib> #include <ctime>

using namespace std;

I include cstdlib because I plan to generate a random number. I include ctime

because I want to seed the random number generator with the current time.

Introducing Guess My Number 75



**Figure 2.15**

The game loop applied to Guess My Number.

Next, I start the main() function by picking a random number, setting the number of tries to 0, and establishing a variable for the player’s guess:

int main()

{

srand(static\_cast<unsigned int>(time(0))); //seed random number generator

int secretNumber = rand() % 100 + 1; // random number between 1 and 100 int tries = 0;

int guess;

cout << "\tWelcome to Guess My Number\n\n";

76 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

#### [Creating the Game Loop](#_bookmark4)

Next, I write the game loop.

do

{

cout << "Enter a guess: "; cin >> guess;

++tries;

if (guess > secretNumber)

{

cout << "Too high!\n\n";

}

else if (guess < secretNumber)

{

}

else

{

}

cout << "Too low!\n\n";

cout << "\nThat’s it! You got it in " << tries << " guesses!\n";

} while (guess != secretNumber);

I get the player’s guess, increment the number of tries, and then tell the player if his guess is too high, too low, or right on the money. If the player’s guess is correct, the loop ends. Notice that the if statements are nested inside the while loop.

#### [Wrapping Up the Game](#_bookmark4)

Once the player has guessed the secret number, the loop and game are over. All that’s left to do is end the program.

return 0;

}

### [Summary](#_bookmark4)

In this chapter, you learned the following concepts:

n You can use the truth or falsity of an expression to branch to (or skip) sections of code.

Summary 77

n You can represent truth or falsity with the keywords, true and false.

n You can evaluate any value or expression for truth or falsity.

n Any non-zero value can be interpreted as true, while 0 can be inter- preted as false.

n A common way to create an expression to be evaluated as true or false

is to compare values with the relational operators.

n The if statement tests an expression and executes a section of code only if the expression is true.

n The else clause of an if statement specifies code that should be executed only if the expression tested in the if statement is false.

n The switch statement tests a value that can be treated as an int and exe- cutes a section of code labeled with the corresponding value.

n The default keyword, when used in a switch statement, specifies code to be executed if the value tested in the switch statement matches no listed values.

n The while loop executes a section of code if an expression is true and repeats the code as long as the expression is true.

n A do loop executes a section of code and then repeats the code as long as the expression is true.

n Used in a loop, the break statement immediately ends the loop.

n Used in a loop, the continue statement immediately causes the control of the program to branch to the top of the loop.

n The && (AND) operator combines two simpler expressions to create a new expression that is true only if both simpler expressions are true.

n The || (OR) operator combines two simpler expressions to create a new expression that is true if either simpler expression is true.

n The ! (NOT) operator creates a new expression that is the opposite truth value of the original.

n The game loop is a generalized representation of the flow of events in a game, the core of which repeats.

78 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

n The file cstdlib contains functions that deal with generating random numbers.

n The function srand(), defined in cstdlib, seeds the random number generator.

n The function rand(), defined in cstdlib, returns a random number.

### [Questions and Answers](#_bookmark4)

Q: Do you have to use the keywords true and false?

A: No, but it’s a good idea. Before the advent of the keywords true and false, programmers often used 1 to represent true and 0 to represent false. However, now that true and false are available, it’s best to use them instead of the old-fashioned 1 and 0.

Q: Can you assign a bool variable something other than true or false?

A: Yes. You can assign an expression to a bool variable, which will store the truth or falsity of the expression.

Q: Can you use a switch statement to test some non-integer value?

A: No. switch statements only work with values that can be interpreted as integers (including char values).

Q: How can you test a single non-integer value against multiple values if you can’t use a switch statement?

A: You can use a series of if statements. Q: What’s an infinite loop?

A: A loop that will never end, regardless of user input. Q: Why are infinite loops considered bad?

A: Because a program stuck in an infinite loop will never end on its own. It has to be shut down by the operating system. In the worst case, a user will have to shut his computer off to end a program stuck in an infinite loop.

Q: Won’t a compiler catch an infinite loop and flag it as an error?

A: No. An infinite loop is a logical error—the kind of error a programmer must track down.

Questions and Answers 79

Q: If infinite loops are a bad thing, then isn’t a while (true) loop a bad thing?

A: No. When a programmer creates a while (true) loop, he should provide a way for the loop to end (usually through a break statement).

Q: Why would a programmer create a while (true) loop?

A: while (true) loops are often used for the main loop of a program, like the game loop.

Q: Why do some people feel that using a break statement to exit a loop is poor programming?

A: Because indiscriminate use of break statements can make it hard to under- stand the conditions under which a loop ends. However, sometimes the use of a while (true) loop along with a break statement can be clearer than creating the same loop in a more traditional way.

Q: What’s a pseudorandom number?

A: A random number that’s usually generated by a formula. As a result, a series of pseudorandom numbers is not truly random, but good enough for most purposes.

Q: What’s seeding a random number generator?

A: It’s giving the random number generator a seed, such as an integer, which affects the way the generator produces random numbers. If you don’t seed a random number generator, it will produce the same series of numbers each time its run from the beginning of a program.

Q: Don’t you always want to seed the random number generator before using it?

A: Not necessarily. You might want a program to produce the exact same sequence of “random” numbers each time it runs for testing purposes, for example.

Q: How can I generate more truly random numbers?

A: There are third-party libraries that produce better pseudorandom numbers than the ones that typically come with C++ compilers.

80 Chapter 2 n Truth, Branching, and the Game Loop: Guess My Number

Q: Do all games use the game loop?

A: The game loop is just a way of looking at a typical game’s flow of events. And just because this paradigm fits a particular game, that doesn’t necessarily mean that the game is implemented with a loop around the bulk of its code.

### [Discussion Questions](#_bookmark4)

1. What kinds of things would be difficult to program without loops?
2. What are the advantages and disadvantages of the switch statement ver- sus a series of if statements?
3. When might you omit a break statement from the end of a case in a

switch statement?

1. When should you use a while loop over a do loop?
2. Describe your favorite game in terms of the game loop. Is the game loop a good fit?

### [Exercises](#_bookmark4)

1. Rewrite the Menu Chooser program from this chapter using an enumera- tion to represent difficulty levels. The variable choice will still be of type int.
2. What’s wrong with the following loop?

int x = 0; while (x)

{

++x;

cout << x << endl;

}

1. Write a new version of the Guess My Number program in which the player and the computer switch roles. That is, the player picks a number and the computer must guess what it is.

|  |  |
| --- | --- |
|  |  |
| [chapter 3](#_bookmark4) | |

# [For Loops, Strings, and Arrays: Word Jumble](#_bookmark4)

You’ve seen how to work with single values, but in this chapter you’ll learn how to work with sequences of data. You’ll learn more about strings—objects for sequences of characters. You’ll also see how to work with sequences of any type. And you’ll discover a new type of loop that’s perfect for use with these sequences. Specifically, you’ll learn to:

n Use for loops to iterate over sequences

n Use objects, which combine data and functions

n Use string objects and their member functions to work with sequences of characters

n Use arrays to store, access, and manipulate sequences of any type

n Use multidimensional arrays to better represent certain collections of data

### [Using for Loops](#_bookmark4)

You met one type of loop in Chapter 2—the while loop. Well, it’s time to meet another—the for loop. Like its cousin the while loop, the for loop lets you repeat a section of code, but for loops are particularly suited for counting and moving through a sequence of things (like the items in an RPG character’s inventory).

81

82 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

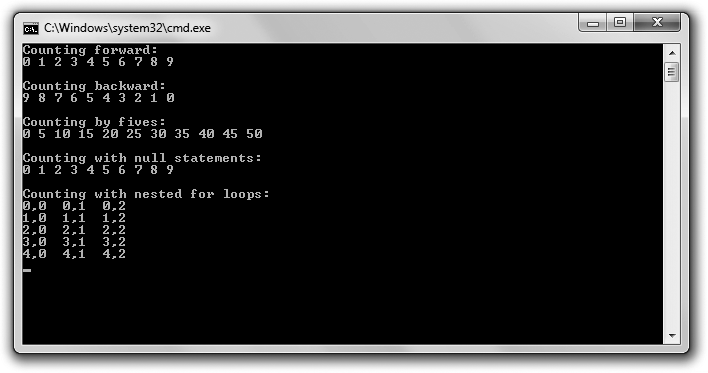
Here’s the generic form of for loop:

for (*initialization*; *test*; *action*) *statement*;

*initialization* is a statement that sets up some initial condition for the loop. (For example, it might set a counter variable to 0.) The expression *test* is tested each time before the loop body executes, just as in a while loop. If *test* is false, the program moves on to the statement after the loop. If *test* is true, the program executes *statement*. Next, *action* is executed (which often involves incrementing a counter variable). The cycle repeats until *test* is false, at which point the loop ends.

#### [Introducing the Counter Program](#_bookmark4)

The Counter program counts forward, backward, and by fives. It even counts out a grid with rows and columns. It accomplishes all of this through the use of for loops. Figure 3.1 shows the program in action.



**Figure 3.1**

for loops do all of the counting, while a pair of nested for loops displays the grid.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 3 folder; the filename is counter.cpp.

Using for Loops 83

// Counter

// Demonstrates for loops #include <iostream> using namespace std;

int main()

{

cout << "Counting forward:\n"; for (int i = 0; i < 10; ++i)

{

cout << i << " ";

}

cout << "\n\nCounting backward:\n"; for (int i = 9; i >= 0; --i)

{

cout << i << " ";

}

cout << "\n\nCounting by fives:\n"; for (int i = 0; i <= 50; i += 5)

{

cout << i << " ";

}

cout << "\n\nCounting with null statements:\n"; int count = 0;

for ( ; count < 10; )

{

cout << count << " ";

++count;

}

cout << "\n\nCounting with nested for loops:\n"; const int ROWS = 5;

const int COLUMNS = 3;

for (int i = 0; i < ROWS; ++i)

{

84 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

for (int j = 0; j < COLUMNS; ++j)

{

cout << i << "," << j << " ";

}

cout << endl;

}

return 0;

}

**Trap**

If you’re using an older compiler that doesn’t fully implement the current C++ standard, when you try to compile this program, you might get an error that says something like: error: ’i’ : redefinition; multiple initialization.

The best solution is to use a modern, compliant compiler. Luckily, you can download the popular (and free) Microsoft Visual C++ Express Edition IDE, which includes a modern compiler, from [http://www.microsoft.com/express/.](http://www.microsoft.com/express/)

If you must use your old compiler, you should declare any for loop counter variables just once for all for loops in a scope. I cover the topic of scopes in Chapter 5, “Functions: Mad Lib.”

#### [Counting with for Loops](#_bookmark4)

The first for loop counts from 0 to 9. The loop begins:

for (int i = 0; i < 10; ++i)

The initialization statement, int i = 0, declares i and initializes it to 0. The expression i < 10 says that the loop will continue as long as i is less than 10. Lastly, the action statement, ++i, says i is to be incremented each time the loop body finishes. As a result, the loop *iterates* 10 times—once for each of the values 0 through 9. And during each iteration, the loop body displays the value of i.

The next for loop counts from 9 down to 0. The loop begins:

for (int i = 9; i >= 0; --i)

Here, i is initialized to 9, and the loop continues as long as i is greater than or equal to 0. Each time the loop body finishes, i is decremented. As a result, the loop displays the values 9 through 0.

Using for Loops 85

The next loop counts from 0 to 50, by fives. The loop begins:

for (int i = 0; i <= 50; i += 5)

Here, i is initialized to 0, and the loop continues as long as i is less than or equal to 50. But notice the action statement, i += 5. This statement increases i by five each time the loop body finishes. As a result, the loop displays the values 0, 5, 10, 15, and so on. The expression i <= 50 says to execute the loop body as long as i is less than or equal to 50.

You can initialize a counter variable, create a test condition, and update the counter variable with any values you want. However, the most common thing to do is to start the counter at 0 and increment it by 1 after each loop iteration.

Finally, the caveats regarding infinite loops that you learned about while studying while loops apply equally well to for loops. Make sure you create loops that can end; otherwise, you’ll have a very unhappy gamer on your hands.

#### [Using Empty Statements in for Loops](#_bookmark4)

You can use empty statements in creating your for loop, as I did in the following loop:

for ( ; count < 10; )

I used an empty statement for the initialization and action statements. That’s fine because I declared and initialized count before the loop and incremented it inside the loop body. This loop displays the same sequence of integers as the very first loop in the program. Although the loop might look odd, it’s perfectly legal.

**Hint**

Different game programmers have different traditions. In the last chapter, you saw that you can create a loop that continues until it reaches an exit statement—such as a break—using while (true). Well, some programmers prefer to create these kinds of loops using a for statement that begins for (;;). Because the test expression in this loop is the empty statement, the loop will continue until it encounters some exit statement.

86 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

#### [Nesting for Loops](#_bookmark4)

You can nest for loops by putting one inside the other. That’s what I did in the following section of code, which counts out the elements of a grid. The outer loop, which begins:

for (int i = 0; i < ROWS; ++i)

simply executes its loop body ROWS (five) times. But it just so happens that there’s another for loop inside this loop, which begins:

for (int j = 0; j < COLUMNS; ++j)

As a result, the inner loop executes in full for each iteration of the outer loop. In this case, that means the inner loop executes COLUMNS (three) times, for the ROWS (five) times the outer loop iterates, for a total of 15 times. Specifically, here’s what happens:

1. The outer for loop declares i and initializes it to 0. Since i is less than

ROWS (5), the program enters the outer loop’s body.

1. The inner loop declares j and initializes it to 0. Since j is less than COLUMNS (3), the program enters its loop body, sending the values of i and j to cout, which displays 0, 0.
2. The program reaches the end of the body of the inner loop and increments j to 1. Since j is still less than COLUMNS (3), the program executes the inner loop’s body again, displaying 0, 1.
3. The program reaches the end of the inner loop’s body and increments j to 2. Since j is still less than COLUMNS (3), the program executes the inner loop’s body again, displaying 0, 2.
4. The program reaches the end of the inner loop’s body and increments j to 3. This time, however, j is not less than COLUMNS (3) and the inner loop ends.
5. The program finishes the first iteration of the outer loop by sending endl

to cout, ending the first row.

1. The program reaches the end of the outer loop’s body and increments i to 1. Since i is less than ROWS (5), the program enters the outer loop’s body again.

Understanding Objects 87

1. The program reaches the inner loop, which starts from the beginning once again, by declaring and initializing j to 0. The program goes through the process I described in Steps 2 through 7, displaying the sec- ond row of the grid. This process continues until all five rows have been displayed.

Again, the important thing to remember is that the inner loop is executed in full for each iteration of the outer loop.

### [Understanding Objects](#_bookmark4)

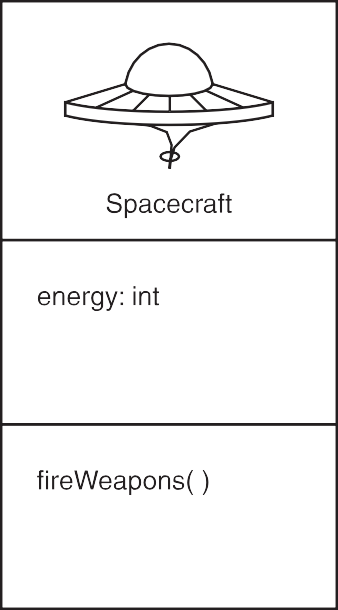
So far you’ve seen how to store individual pieces of information in variables and how to manipulate those variables using operators and functions. But most of the things you want to represent in games—such as, say, an alien spacecraft— are objects. They’re encapsulated, cohesive things that combine qualities (such as an energy level) and abilities (for example, firing weapons). Often it makes no sense to talk about the individual qualities and abilities in isolation from each other.

Fortunately, most modern programming languages let you work with software objects (often just called *objects*) that combine data and functions. A data element of an object is called a *data member*, while a function of an object is called a *member function*. As a concrete example, think about that alien spacecraft. An alien spacecraft object might be of a new type called Spacecraft, defined by a game programmer, and might have a data member for its energy level and a member function to fire its weapons. In practice, an object’s energy level might be stored in its data member energy as an int, and its ability to fire

its weapons might be defined in a member function called fireWeapons().

Every object of the same type has the same basic structure, so each object will have the same set of data members and member functions. However, as an individual, each object will have its own values for its data members. If you had a squadron of five alien spacecrafts, each would have its own energy level. One might have an energy level of 75, while another might have a level of only 10, and so on. Even if two crafts have the same energy level, each would belong to a unique spacecraft. Each craft could also fire its own weapons with a call to its member function, fireWeapons(). Figure 3.2 illustrates the concept of an alien spacecraft.

88 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble



**Figure 3.2**

This representation of the definition of an alien spacecraft says that each object will have a data member called energy and a member function called fireWeapons().

The cool thing about objects is that you don’t need to know the implementation details to use them—just as you don’t need to know how to build a car in order to drive one. You only have to know the object’s data members and member functions—just as you only need to know where a car’s steering wheel, gas pedal, and brake pedal are located.

You can store objects in variables, just like with built-in types. Therefore, you could store an alien spacecraft object in a variable of the Spacecraft type. You can access data members and member functions using the member selection operator (.), by placing the operator after the variable name of the object. So if you want your alien spacecraft, ship, to fire its weapons only if its energy level is greater than 10, you could write:

// ship is an object of Spacecraft type if (ship.energy > 10)

{

ship.fireWeapons()

}

Using String Objects 89

ship.energy accesses the object’s energy data member, while ship.fireWeapons()

calls the object’s fireWeapons() member function.

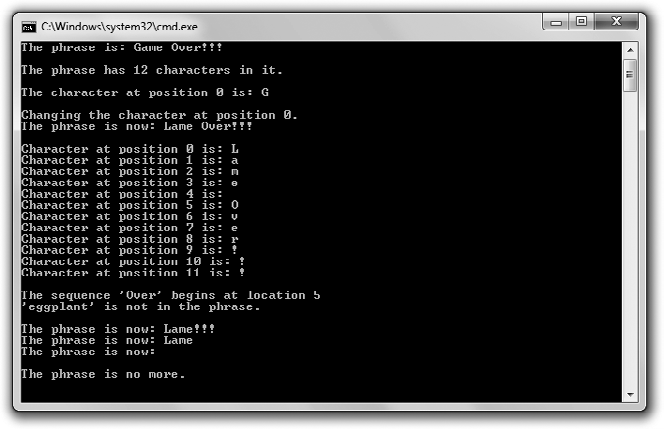
Although you can’t make your own new types (like for an alien spacecraft) just yet, you can work with previously defined object types. And that’s next on the agenda.

### [Using String Objects](#_bookmark4)

string objects, which you met briefly in Chapter 1, are the perfect way to work with sequences of characters, whether you’re writing a complete word puzzle game or simply storing a player’s name. A string is actually an object, and it provides its own set of member functions that allow you to do a range of things with the string object—everything from simply getting its length to performing complex character substitutions. In addition, strings are defined so that they work intuitively with a few of the operators you already know.

#### [Introducing the String Tester Program](#_bookmark4)

The String Tester program uses the string object equal to "Game Over!!!" and tells you its length, the index (position number) of each character, and whether or not certain substrings can be found in it. In addition, the program erases parts of the string object. Figure 3.3 shows the results of the program.



**Figure 3.3**

String objects are combined, changed, and erased through familiar operators and string member functions.

90 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 3 folder; the filename is string\_tester.cpp.

// String Tester

// Demonstrates string objects

#include <iostream> #include <string>

using namespace std;

int main()

{

string word1 = "Game"; string word2("Over"); string word3(3, ’!’);

string phrase = word1 + " " + word2 + word3; cout << "The phrase is: " << phrase << "\n\n";

cout << "The phrase has " << phrase.size() << " characters in it.\n\n"; cout << "The character at position 0 is: " << phrase[0] << "\n\n";

cout << "Changing the character at position 0.\n"; phrase[0] = ’L’;

cout << "The phrase is now: " << phrase << "\n\n";

for (unsigned int i = 0; i < phrase.size(); ++i)

{

cout << "Character at position " << i << " is: " << phrase[i] << endl;

}

cout << "\nThe sequence ’Over’ begins at location "; cout << phrase.find("Over") << endl;

Using String Objects 91

if (phrase.find("eggplant") == string::npos)

{

cout << "’eggplant’ is not in the phrase.\n\n";

}

phrase.erase(4, 5);

cout << "The phrase is now: " << phrase << endl;

phrase.erase(4);

cout << "The phrase is now: " << phrase << endl;

phrase.erase();

cout << "The phrase is now: " << phrase << endl;

if (phrase.empty())

{

cout << "\nThe phrase is no more.\n";

}

return 0;

}

#### [Creating string Objects](#_bookmark4)

The first thing I do in main() is create three strings in three different ways:

string word1 = "Game"; string word2("Over"); string word3(3, ’!’);

In the first line of this group, I simply create the string object word1 using the assignment operator, the same way you’ve seen for other variables. As a result, word1 is "Game".

Next, I create word2 by placing the string object to which I want the variable set between a pair of parentheses. As a result, word2 is "Over".

Finally, I create word3 by supplying between a pair of parentheses a number followed by a single character. This produces a string object made up of the provided character, which has a length equal to the number. As a result, word3 is "!!!".

92 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

#### [Concatenating string Objects](#_bookmark4)

Next, I create a new string object, phrase, by concatenating the first three

string objects:

string phrase = word1 + " " + word2 + word3;

As a result, phrase is "Game Over!!!".

Notice that the + operator, which you’ve seen work only with numbers, also concatenates string objects. That’s because the + operator has been overloaded. Now, when you first hear the term *overloaded*, you might think it’s a bad thing—the operator is about to blow! But it’s a good thing. Operator over- loading redefines a familiar operator so it works differently when used in a new, previously undefined context. In this case, I use the + operator not to add numbers, but to join string objects. I’m able to do this only because the string type specifically overloads the + operator and defines it so the operator means string object concatenation when used with strings.

#### [Using the size() Member Function](#_bookmark4)

Okay, it’s time to take a look at a string member function. Next, I use the member function size():

cout << "The phrase has " << phrase.size() << " characters in it.\n\n";

phrase.size() calls the member function size() of the string object phrase through the member selection operator . (the dot). The size() member function simply returns an unsigned integer value of the size of the string object—its number of characters. Because the string object is "Lame Over!!!", the member function returns 12. (Every character counts, including spaces.) Of course, calling size() for another string object might return a different result based on the number of characters in the string object.

**Hi n t**

string objects also have a member function length(), which, just like size(), returns the number of characters in the string object.

Using String Objects 93

#### [Indexing a string Object](#_bookmark4)

A string object stores a sequence of char values. You can access any individual char value by providing an index number with the subscripting operator ([]). That’s what I do next:

cout << "The character at position 0 is: " << phrase[0] << "\n\n";

The first element in a sequence is at position 0. In the previous statement, phrase

[0] is the character G. And because counting begins at 0, the last character in the string object is phrase[11], even though the string object has 12 characters  in it.

**Trap**

It’s a common mistake to forget that indexing begins at position 0. Remember, a string object with *n* characters in it can be indexed from position 0 to position n - 1.

Not only can you access characters in a string object with the subscripting operator, but you can also reassign them. That’s what I do next:

phrase[0] = ’L’;

I change the first character of phrase to the character L, which means phrase

becomes "Lame Over!!!"

**Trap**

Cþþ compilers do not perform bounds checking when working with string objects and the subscripting operator. This means that the compiler doesn’t check to see whether you’re attempting to access an element that doesn’t exist. Accessing an invalid sequence element can lead to disastrous results because it’s possible to write over critical data in your computer’s memory. By doing this, you can crash your program, so take care when using the subscripting operator.

#### [Iterating through string Objects](#_bookmark4)

Given your new knowledge of for loops and string objects, it’s a snap to iterate through the individual characters of a string object. That’s what I do next:

for (unsigned int i = 0; i < phrase.size(); ++i)

{

cout << "Character at position " << i << " is: " << phrase[i] << endl;

}

94 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

The loop iterates through all of the valid positions of phrase. It starts with 0 and goes through 11. During each iteration, a character of the string object is displayed with phrase[i]. Note that I made the loop variable, i, an unsigned int because the value returned by size() is an unsigned integral type.

**Real World**

Iterating through a sequence is a powerful and often-used technique in games. You might, for example, iterate through hundreds of individual units in a strategy game, updating their status and order. Or you might iterate through the list of vertices of a 3D model to apply some geometric transformation.

#### [Using the find() Member Function](#_bookmark4)

Next I use the member function find() to check whether either of two string literals are contained in phrase. First, I check for the string literal "Over":

cout << "\nThe sequence ’Over’ begins at location "; cout << phrase.find("Over") << endl;

The find() member function searches the calling string object for the string supplied as an argument. The member function returns the position number of the first occurrence where the string object for which you are searching begins in the calling string object. This means that phrase.find("Over") returns the position number where the first occurrence of "Over" begins in phrase. Since phrase is "Lame Over!!!", find() returns 5. (Remember, position numbers begin at 0, so 5 means the sixth character.)

But what if the string for which you are searching doesn’t exist in the calling string? I tackle that situation next:

if (phrase.find("eggplant") == string::npos)

{

cout << "’eggplant’ is not in the phrase.\n\n";

}

Because "eggplant" does not exist in phrase, find() returns a special constant defined in the file string, which I access with string::npos. As a result, the screen displays the message, “’eggplant’ is not in the phrase.”

Using String Objects 95

The constant I access through string::npos represents the largest possible size of a string object, so it is greater than any possible valid position number in a string object. Informally, it means “a position number that can’t exist.” It’s the perfect return value to indicate that one string couldn’t be found in another.

**Hint**

When using find(), you can supply an optional argument that specifies a character number for the program to start looking for the substring. The following line will start looking for the string literal "eggplant" beginning at position 5 in the string object phrase.

location = phrase.find("eggplant", 5);

#### [Using the erase() Member Function](#_bookmark4)

The erase() member function removes a specified substring from a string object. One way to call the member function is to specify the beginning position and the length of the substring, as I did in this code:

phrase.erase(4, 5);

The previous line removes the five-character substring starting at position 4. Because phrase is "Lame Over!!!", the member function removes the substring Over and, as a result, phrase becomes "Lame!!!".

Another way to call erase() is to supply just the beginning position of the substring. This removes all of the characters starting at that position number to the end of the string object. That’s what I do next:

phrase.erase(4);

This line removes all of the characters of the string object starting at position 4. Since phrase is "Lame!!!", the member function removes the substring !!! and, as a result, phrase becomes "Lame".

Yet another way to call erase() is to supply no arguments, as I did in this code:

phrase.erase();

96 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

The previous line erases every character in phrase. As a result, phrase becomes the empty string, which is equal to "".

#### [Using the empty() Member Function](#_bookmark4)

The empty() member function returns a bool value—true if the string object is empty and false otherwise. I use empty() in the following code:

if (phrase.empty())

{

cout << "\nThe phrase is no more.\n";

}

Because phrase is equal to the empty string, phrase().empty returns true, and the screen displays the message, “The phrase is no more.”

### [Using Arrays](#_bookmark4)

While string objects provide a great way to work with a sequence of characters, arrays provide a way to work with elements of any type. That means you can use an array to store a sequence of integers for, say, a high-score list. But it also means that you can use arrays to store elements of programmer-defined types, such as a sequence of items that an RPG character might carry.

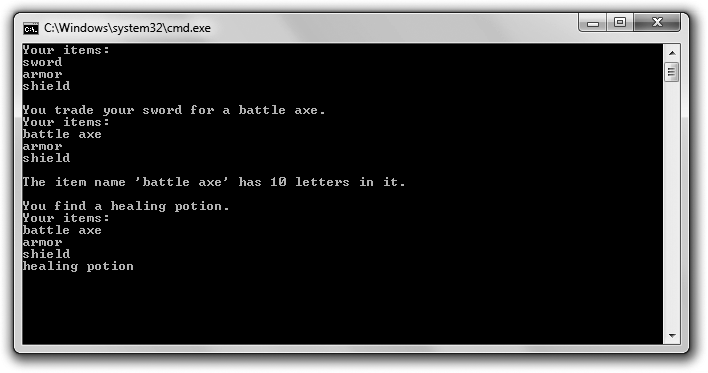
#### [Introducing the Hero’s Inventory Program](#_bookmark4)

The Hero’s Inventory program maintains the inventory of a hero from a typical RPG. Like in most RPGs, the hero is from a small, insignificant village, and his father was killed by an evil warlord. (What’s a quest without a dead father?) Now that the hero has come of age, it’s time for him to seek his revenge.

In this program, the hero’s inventory is represented by an array. The array is a sequence of string objects—one for each item in the hero’s possession. The hero trades and even finds new items. Figure 3.4 shows the program in action.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 3 folder; the filename is heros\_inventory.cpp.

Using Arrays 97



**Figure 3.4**

The hero’s inventory is a sequence of string objects stored in an array.

// Hero’s Inventory

// Demonstrates arrays

#include <iostream> #include <string>

using namespace std; int main()

{

const int MAX\_ITEMS = 10; string inventory[MAX\_ITEMS];

int numItems = 0; inventory[numItems++] = "sword"; inventory[numItems++] = "armor"; inventory[numItems++] = "shield";

cout << "Your items:\n";

for (int i = 0; i < numItems; ++i)

{

cout << inventory[i] << endl;

}

98 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

cout << "\nYou trade your sword for a battle axe."; inventory[0] = "battle axe";

cout << "\nYour items:\n";

for (int i = 0; i < numItems; ++i)

{

cout << inventory[i] << endl;

}

cout << "\nThe item name ’" << inventory[0] << "’ has "; cout << inventory[0].size() << " letters in it.\n";

cout << "\nYou find a healing potion."; if (numItems < MAX\_ITEMS)

{

inventory[numItems++] = "healing potion";

}

else

{

cout << "You have too many items and can’t carry another.";

}

cout << "\nYour items:\n";

for (int i = 0; i < numItems; ++i)

{

cout << inventory[i] << endl;

}

return 0;

}

#### [Creating Arrays](#_bookmark4)

It’s often a good idea to define a constant for the number of elements in an array. That’s what I did with MAX\_ITEMS, which represents the maximum number of items the hero can carry:

const int MAX\_ITEMS = 10;

You declare an array much the same way you would declare any variable you’ve seen so far: You provide a type followed by a name. In addition, your compiler must know the size of the array so it can reserve the necessary memory space. You can provide that information following the array name, surrounded by square brackets. Here’s how I declare the array for the hero’s inventory:

string inventory[MAX\_ITEMS];

Using Arrays 99

The preceding code declares an array inventory of MAX\_ITEMS string objects. (Because MAX\_ITEMS is 10, that means 10 string objects.)

**Trick**

You can initialize an array with values when you declare it by providing an *initializer list*—a sequence of elements separated by commas and surrounded by curly braces. Here’s an example:

string inventory[MAX\_ITEMS] = {"sword", "armor", "shield"};

The preceding code declares an array of string objects, inventory, that has a size of MAX\_ITEMS. The first three elements of the array are initialized to "sword", "armor", and "shield".

If you omit the number of elements when using an initializer list, the array will be created with a size equal to the number of elements in the list. Here’s an example:

string inventory[] = {"sword", "armor", "shield"};

Because there are three elements in the initializer list, the preceding line creates an array, inventory, that is three elements in size. Its elements are "sword", "armor", and "shield".

#### [Indexing Arrays](#_bookmark4)

You index arrays much like you index string objects. You can access any individual element by providing an index number with the subscripting operator ([]).

Next, I add three items to the hero’s inventory using the subscripting operator:

int numItems = 0; inventory[numItems++] = "sword"; inventory[numItems++] = "armor"; inventory[numItems++] = "shield";

I start by defining numItems for the number of items the hero is carrying at the moment. Next I assign "sword" to position 0 of the array. Because I use the postfix increment operator, numItems is incremented after the assignment to the array. The next two lines add "armor" and "shield" to the array, leaving numItems at the correct value of 3 when the code finishes.

Now that the hero is stocked with some items, I display his inventory:

cout << "Your items:\n";

for (int i = 0; i < numItems; ++i)

100 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

{

cout << inventory[i] << endl;

}

This should remind you of string indexing. The code loops through the first three elements of inventory, displaying each string object in order.

Next, the hero trades his sword for a battle axe. I accomplish this through the following line:

inventory[0] = "battle axe";

The previous code reassigns the element at position 0 in inventory with the string object "battle axe". Now the first three elements of inventory are "battle axe", "armor", and "shield".

**Trap**

Array indexing begins at 0, just as you saw with string objects. This means that the following code defines a five-element array:

int highScores[5];

Valid position numbers are 0 through 4, inclusive. There is no element highScores[5]! An attempt to access highScores[5] could lead to disastrous results, including a program crash.

#### [Accessing Member Functions of an Array Element](#_bookmark4)

You can access the member functions of an array element by writing the array element, followed by the member selection operator, followed by the member function name. This sounds a bit complicated, but it’s not. Here’s an example:

cout << inventory[0].size() << " letters in it.\n";

The code inventory[0].size() means the program should call the size() member function of the element inventory[0]. In this case, because inventory[0] is "battle axe", the call returns 10, the number of characters in the string object.

#### [Being Aware of Array Bounds](#_bookmark4)

As you learned, you have to be careful when you index an array. Because an array has a fixed size, you can create an integer constant to store the size of an array. Again, that’s just what I did in the beginning of the program:

const int MAX\_ITEMS = 10;

Understanding C-Style Strings 101

In the following lines, I use MAX\_ITEMS to protect myself before adding another item to the hero’s inventory:

if (numItems < MAX\_ITEMS)

{

}

else

{

}

inventory[numItems++] = "healing potion";

cout << "You have too many items and can’t carry another.";

In the preceding code, I first checked to see whether numItems is less than MAX\_ITEMS. If it is, then I can safely use numItems as an index and assign a new string object to the array. In this case numItems is 3, so I assign the string "healing potion" to array position 3. If this hadn’t been the case, then I would have displayed the message, “You have too many items and can’t carry another.”

So what happens if you do attempt to access an array element outside the bounds of the array? It depends, because you’d be accessing some unknown part of the computer’s memory. At worst, if you attempt to assign some value to an element outside the bounds of an array you could cause your program to do unpredictable things and it might even crash.

Testing to make sure that an index number is a valid array position before using it is called *bounds checking*. It’s critical for you to perform bounds checking when there’s a chance that an index you want to use might not be valid.

### [Understanding C-Style Strings](#_bookmark4)

Before string objects came along, C++ programmers represented strings with arrays of characters terminated by a null character. These arrays of characters are now called *C-style strings* because the practice began in C programs. You can declare and initialize a C-style string like you would any other array:

char phrase[] = "Game Over!!!";

C-style strings terminate with a character called the *null character* to signify their end. You can write the null character as ’\0’. I didn’t need to use the null character in the previous code because it is stored at the end of the string for me. So technically, phrase has 13 elements. (However, functions that work with

102 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

C-style strings will say that phrase has a length of 12, which makes sense and is in line with how string objects work.)

As with any other type of array, you can specify the array size when you define it. So another way to declare and initialize a C-style string is

char phrase[81] = "Game Over!!!";

The previous code creates a C-style string that can hold 80 printable characters (plus its terminating null character).

C-style strings don’t have member functions. But the cstring file, which is part of the standard library, contains a variety of functions for working with C-style strings.

A nice thing about string objects is that they’re designed to work seamlessly with C-style strings. For example, all of the following are completely valid uses of C-style strings with string objects:

string word1 = "Game"; char word2[] = " Over";

string phrase = word1 + word2; if (word1 != word2)

{

cout << "word1 and word2 are not equal.\n";

}

if (phrase.find(word2) != string::npos)

{

cout << "word2 is contained in phrase.\n";

}

You can concatenate string objects and C-style strings, but the result is always a string object (so the code char phrase2[] = word1 + word2; would produce an error). You can compare string objects and C-style strings using the relational operators. And you can even use C-style strings as arguments in string object member functions.

C-style strings have the same shortcomings as arrays. One of the biggest is that their lengths are fixed. So the moral is: Use string objects whenever possible, but be prepared to work with C-style strings if necessary.

Using Multidimensional Arrays 103

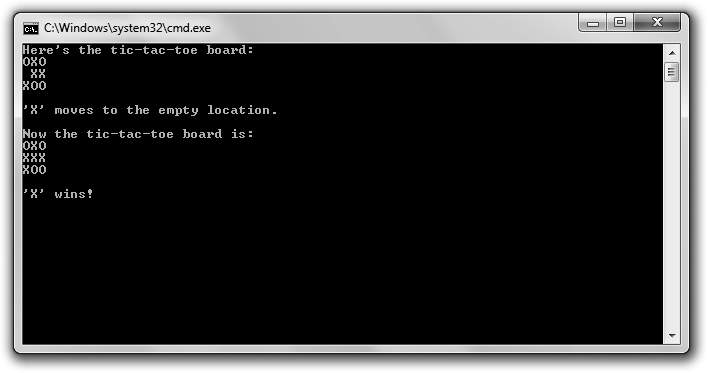
### [Using Multidimensional Arrays](#_bookmark4)

As you’ve seen, sequences are great for games. You can use them in the form of a string to store a player’s name, or you can use them in the form of an array to store a list of items in an RPG. But sometimes part of a game cries out for more than a linear list of things. Sometimes part of a game literally requires more dimension. For example, while you could represent a chessboard with a 64-element array, it really is much more intuitive to work with it as a two- dimensional entity of 8 x 8 elements. Fortunately, you can create an array of two or three (or even more) dimensions to best fit your game’s needs.

#### [Introducing the Tic-Tac-Toe Board Program](#_bookmark4)

The Tic-Tac-Toe Board program displays a tic-tac-toe board. The program displays the board and declares X the winner. Although the program could have been written using a one-dimensional array, it uses a two-dimensional array to represent the board. Figure 3.5 illustrates the program.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 3 folder; the filename is tic-tac-toe\_board.cpp.



**Figure 3.5**

The tic-tac-toe board is represented by a two-dimensional array.

104 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

// Tic-Tac-Toe Board

// Demonstrates multidimensional arrays #include <iostream>

using namespace std;

int main()

{

const int ROWS = 3; const int COLUMNS = 3;

char board[ROWS][COLUMNS] = { {’O’, ’X’, ’O’},

{’ ’, ’X’, ’X’},

{’X’, ’O’, ’O’} };

cout << "Here’s the tic-tac-toe board:\n"; for (int i = 0; i < ROWS; ++i)

{

for (int j = 0; j < COLUMNS; ++j)

{

cout << board[i][j];

}

cout << endl;

}

cout << "\n’X’ moves to the empty location.\n\n"; board[1][0] = ’X’;

cout << "Now the tic-tac-toe board is:\n"; for (int i = 0; i < ROWS; ++i)

{

for (int j = 0; j < COLUMNS; ++j)

{

cout << board[i][j];

}

cout << endl;

}

Using Multidimensional Arrays 105

cout << "\n’X’ wins!";

return 0;

}

#### [Creating Multidimensional Arrays](#_bookmark4)

One of the first things I do in the program is declare and initialize an array for the tic-tac-toe board.

char board[ROWS][COLUMNS] = { {’O’, ’X’, ’O’},

{’ ’, ’X’, ’X’},

{’X’, ’O’, ’O’} };

The preceding code declares a 3 x 3 (since ROWS and COLUMNS are both 3) two- dimensional character array. It also initializes all of the elements.

**Hint**

It’s possible to simply declare a multidimensional array without initializing it. Here’s an example:

char chessBoard[8][8];

The preceding code declares an 8 x 8, two-dimensional character array, chessBoard. By the way, multidimensional arrays aren’t required to have the same size for each dimension. The following is a perfectly valid declaration for a game map represented by individual characters:

char map[12][20];

#### [Indexing Multidimensional Arrays](#_bookmark4)

The next thing I do in the program is display the tic-tac-toe board. But before I get into the details of that, I want to explain how to index an individual array element. You index an individual element of a multidimensional array by supplying a value for each dimension of the array. That’s what I do to place an X in the array where a space was:

board[1][0] = ’X’;

The previous code assigns the character to the element at board[1][0] (which was ’ ’). Then I display the tic-tac-toe board after the move the same way I displayed it before the move.

for (int i = 0; i < ROWS; ++i)

{

106 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

for (int j = 0; j < COLUMNS; ++j)

{

cout << board[i][j];

}

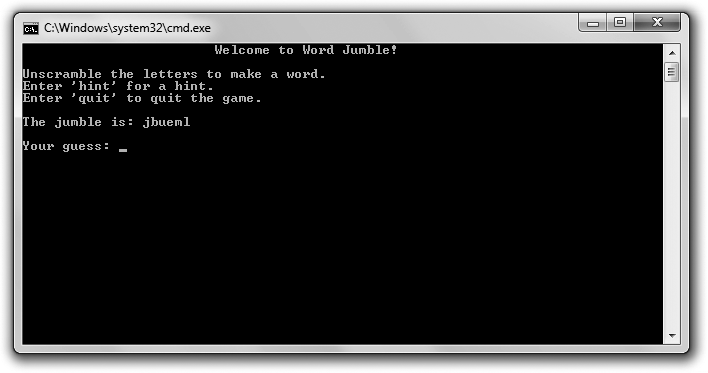
cout << endl;

}

By using a pair of nested for loops, I move through the two-dimensional array and display the character elements as I go, forming a tic-tac-toe board.

### [Introducing Word Jumble](#_bookmark5)

Word Jumble is a puzzle game in which the computer creates a version of a word where the letters are in random order. The player has to guess the word to win the game. If the player is stuck, he or she can ask for a hint. Figure 3.6 shows the game.



**Figure 3.6**

Hmm.. .the word looks “jumbled.”

Introducing Word Jumble 107

**Real World**

Even though puzzle games don’t usually break into the top-ten list of games, major companies still publish them year after year. Why? For one simple reason: They’re profitable. Puzzle games, while not usually blockbusters, can still sell well. There are many gamers out there (casual and hardcore) who are drawn to the Zen of a well-designed puzzle game. And puzzle games cost much less to produce than the high-profile games that require large production teams and years of development time.

#### [Setting Up the Program](#_bookmark5)

As usual, I start with some comments and include the files I need. You can download the code for this program from the Course Technology website [(www.](http://www.courseptr.com/downloads) [courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 3 folder; the filename is word\_jumble.cpp.

// Word Jumble

// The classic word jumble game where the player can ask for a hint

#include <iostream> #include <string> #include <cstdlib> #include <ctime>

using namespace std;

#### [Picking a Word to Jumble](#_bookmark5)

My next task is to pick a word to jumble—the word the player will try to guess. First, I create a list of words and hints:

int main()

{

enum fields {WORD, HINT, NUM\_FIELDS}; const int NUM\_WORDS = 5;

const string WORDS[NUM\_WORDS][NUM\_FIELDS] =

{

{"wall", "Do you feel you’re banging your head against something?"},

{"glasses", "These might help you see the answer."},

{"labored", "Going slowly, is it?"},

{"persistent", "Keep at it."},

{"jumble", "It’s what the game is all about."}

};

108 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

I declare and initialize a two-dimensional array with words and corresponding hints. The enumeration defines enumerators for accessing the array. For example, WORDS[x][WORD] is always a string object that is one of the words, while WORDS[x][HINT] is the corresponding hint.

**Trick**

You can list a final enumerator in an enumeration as a convenient way to store the number of elements. Here’s an example:

enum difficulty {EASY, MEDIUM, HARD, NUM\_DIFF\_LEVELS};

cout << "There are " << NUM\_DIFF\_LEVELS << " difficulty levels.";

In the previous code, NUM\_DIFF\_LEVELS is 3, the exact number of difficulty levels in the enumeration. As a result, the second line of code displays the message, "There are 3 difficulty levels."

Next, I pick a random word from my choices.

srand(static\_cast<unsigned int>(time(0))); int choice = (rand() % NUM\_WORDS);

string theWord = WORDS[choice][WORD]; // word to guess string theHint = WORDS[choice][HINT]; // hint for word

I generate a random index based on the number of words in the array. Then I assign both the random word at that index and it’s corresponding hint to the variables theWord and theHint.

#### [Jumbling the Word](#_bookmark5)

Now that I have the word for the player to guess, I need to create a jumbled version of it.

string jumble = theWord; // jumbled version of word int length = jumble.size();

for (int i = 0; i < length; ++i)

{

int index1 = (rand() % length); int index2 = (rand() % length); char temp = jumble[index1]; jumble[index1] = jumble[index2]; jumble[index2] = temp;

}

Introducing Word Jumble 109

In the preceding code, I created a copy of the word jumble to.. .well, jumble. I generated two random positions in the string object and swapped the characters at those positions. I did this a number of times equal to the length of the word.

#### [Welcoming the Player](#_bookmark5)

Now it’s time to welcome the player, which is what I do next.

cout << "\t\t\tWelcome to Word Jumble!\n\n";

cout << "Unscramble the letters to make a word.\n"; cout << "Enter ’hint’ for a hint.\n";

cout << "Enter ’quit’ to quit the game.\n\n"; cout << "The jumble is: " << jumble;

string guess;

cout << "\n\nYour guess: "; cin >> guess;

I gave the player instructions on how to play, including how to quit and how to ask for a hint.

**Hint**

As enthralling as you think your game is, you should always provide a way for the player to exit it.

#### [Entering the Game Loop](#_bookmark5)

Next, I enter the game loop.

while ((guess != theWord) && (guess != "quit"))

{

if (guess == "hint")

{

cout << theHint;

}

else

{

cout << "Sorry, that’s not it.";

}

cout <<"\n\nYour guess: ";

110 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

cin >> guess;

}

The loop continues to ask the player for a guess until the player either guesses the word or asks to quit.

#### [Saying Goodbye](#_bookmark5)

When the loop ends, the player has either won or quit, so it’s time to say goodbye.

if (guess == theWord)

{

cout << "\nThat’s it! You guessed it!\n";

}

cout << "\nThanks for playing.\n";

return 0;

}

If the player has guessed the word, I congratulate him or her. Finally, I thank the player for playing.

### [Summary](#_bookmark5)

In this chapter, you learned the following concepts:

n The for loop lets you repeat a section of code. In a for loop, you can provide an initialization statement, an expression to test, and an action to take after each loop iteration.

n for loops are often used for counting or looping through a sequence.

n Objects are encapsulated, cohesive entities that combine data (called *data members*) and functions (called *member functions*).

n string objects (often just called *strings*) are defined in the file string, which is part of the standard library. string objects allow you to store a sequence of characters and also have member functions.

n string objects are defined so that they work intuitively with familiar operators, such as the concatenation operator and the relational operators.

Questions and Answers 111

n All string objects have member functions, including those for determining a string object’s length, determining whether or not a string object is empty, finding substrings, and removing substrings.

n Arrays provide a way to store and access sequences of any type.

n A limitation of arrays is that they have a fixed length.

n You can access individual elements of string objects and arrays through the subscripting operator.

n Bounds checking is not enforced when attempts are made to access individual elements of string objects or arrays. Therefore, bounds checking is up to the programmer.

n C-style strings are character arrays terminated with the null character. They are the standard way to represent strings in the C language. And even though C-style strings are perfectly legal in Cþþ, string objects are the preferred way to work with sequences of characters.

n Multidimensional arrays allow for access to array elements using multiple subscripts. For example, a chessboard can be represented as a two-dimensional array, 8 x 8 elements.

### [Questions and Answers](#_bookmark5)

Q: Which is better, a while loop or a for loop?

A: Neither is inherently better than the other. Use the loop that best fits your needs.

Q: When might it be better to use a for loop than a while loop?

A: You can create a while loop to do the job of any for loop; however, there are some cases that cry out for a for loop. Those include counting and iterating through a sequence.

Q: Can I use break and continue statements with for loops?

A: Sure. And they behave just like they do in while loops: break ends the loop and continue jumps control back to the top of the loop.

Q: Why do programmers tend to use variable names such as i, j, and k as counters in for loops?

112 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

A: Believe it or not, programmers use i, j, and k mainly out of tradition. The practice started in early versions of the FORTRAN language, in which integer variables had to start with certain letters, including i, j, and k.

Q: I don’t have to include a file to use int or char types, so why do I have to include the string file to use strings?

A: int and char are built-in types. They’re always accessible in any Cþþ program. The string type, on the other hand, is not a built-in type. It’s defined as part of the standard library in the file string.

Q: How did C-style strings get their name?

A: In the C programming language, programmers represent strings with arrays of characters terminated by a null character. This practice carried over to Cþþ. After the new string type was introduced in Cþþ, programmers needed a way to differentiate between the two. Therefore, the old method was dubbed C-style strings.

Q: Why should I use string objects instead of C-style strings?

A: string objects have advantages over C-style strings. The most obvious is that they are dynamically sizeable. You don’t have to specify a length limit when you create one.

Q: Should I ever use C-style strings?

A: You should opt for string objects whenever possible. If you’re working on an existing project that uses C-style strings, then you might have to work with C-style strings.

Q: What is operator overloading?

A: It’s a process that allows you to define the use of familiar operators in different contexts with different but predictable results. For example, the + operator that is used to add numbers is overloaded by the string type to join strings.

Q: Can’t operator overloading be confusing?

A: It’s true that by overloading an operator you give it another meaning. But the new meaning applies only in a specific new context. For example, it’s clear in

Discussion Questions 113

the expression 4+6 that the + operator adds numbers, while in the expression

myString1 + myString2, the + operator joins strings. Q: Can I use the += operator to concatenate strings?

A: Yes, the += operator is overloaded so it works with strings.

Q: To get the number of characters in a string object, should I use the length()

member function or the size() member function?

A: Both length() and size() return the same value, so you can use either. Q: What’s a predicate function?

A: A function that returns either true or false. The string object member function empty() is an example of a predicate function.

Q: What happens if I try to assign a value to an element beyond the bounds of an array?

A: Cþþ will allow you to make the assignment. However, the results are unpredictable and might cause your program to crash. That’s because you’re altering some unknown part of your computer’s memory.

Q: Why should I use multidimensional arrays?

A: To make working with a group of elements more intuitive. For example, you could represent a chessboard with a one-dimensional array, as in chessBoard[64], or you could represent it with a more intuitive, two- dimensional array, as in chessBoard[8][8].

### [Discussion Questions](#_bookmark5)

1. What are some of the things from your favorite game that you could represent as objects? What might their data members and member functions be?
2. What are the advantages of using an array over a group of individual variables?
3. What are some limitations imposed by a fixed array size?

114 Chapter 3 n For Loops, Strings, and Arrays: Word Jumble

1. What are the advantages and disadvantages of operator overloading?
2. What kinds of games could you create using string objects, arrays, and

for loops as your main tools?

### [Exercises](#_bookmark5)

1. Improve the Word Jumble game by adding a scoring system. Make the point value for a word based on its length. Deduct points if the player asks for a hint.
2. What’s wrong with the following code?

for (int i = 0; i <= phrase.size(); ++i)

{

cout << "Character at position " << i << " is: " << phrase[i] << endl;

}

1. What’s wrong with the following code?

const int ROWS = 2; const int COLUMNS = 3;

char board[COLUMNS][ROWS] = { {’O’, ’X’, ’O’},

{’ ’, ’X’, ’X’} };

|  |  |
| --- | --- |
|  |  |
| [chapter 4](#_bookmark5) | |

# [The Standard Template Library: Hangman](#_bookmark5)

So far you’ve seen how to work with sequences of values using arrays. But there are more sophisticated ways to work with collections of values. In fact, working with collections is so common that part of standard C++ is dedicated to doing just that. In this chapter, you’ll get an introduction to this important library. Specifically, you’ll learn to:

n Use vector objects to work with sequences of values

n Use vector member functions to manipulate sequence elements

n Use iterators to move through sequences

n Use library algorithms to work with groups of elements

n Plan your programs with pseudocode

[Introducing the Standard Template Library](#_bookmark5) Good game programmers are lazy. It’s not that they don’t want to work; it’s just that they don’t want to redo work that’s already been done—especially if it has

been done well. The STL (*Standard Template Library*) represents a powerful

collection of programming work that’s been done well. It provides a group of containers, algorithms, and iterators, among other things.

So what’s a container and how can it help you write games? Well, containers let you store and access collections of values of the same type. Yes, arrays let you do

115

116 Chapter 4 n The Standard Template Library: Hangman

the same thing, but the STL containers offer more flexibility and power than a simple but trusty array. The STL defines a variety of container types; each works in a different way to meet different needs.

The algorithms defined in the STL work with its containers. The *algorithms* are common functions that game programmers find themselves repeatedly applying to groups of values. They include algorithms for sorting, searching, copying, merging, inserting, and removing container elements. The cool thing is that the same algorithm can work its magic on many different container types.

*Iterators* are objects that identify elements in containers and can be manipulated to move among elements. They’re great for, well, iterating through containers. In addition, iterators are required by the STL algorithms.

All of this makes a lot more sense when you see an actual implementation of one of the container types, so that’s up next.

### [Using Vectors](#_bookmark5)

The vector class defines one kind of container provided by the STL. It meets the general description of a *dynamic array*—an array that can grow and shrink in size as needed. In addition, vector defines member functions to manipulate vector elements. This means that the vector has all of the functionality of the array plus more.

At this point, you may be thinking to yourself: Why learn to use these fancy new vectors when I can already use arrays? Well, vectors have certain advantages over arrays, including:

n Vectors can grow as needed while arrays cannot. This means that if you use a vector to store objects for enemies in a game, the vector will grow to accommodate the number of enemies that get created. If you use an array, you have to create one that can store some maximum number of enemies. And if, during play, you need more room in the array than you thought, you’re out of luck.

n Vectors can be used with the STL algorithms while arrays cannot. This means that by using vectors you get complex functionality like searching and sorting, built-in. If you use arrays, you have to write your own code to achieve this same functionality.

Using Vectors 117

There are a few disadvantages to vectors when compared to arrays, including:

n Vectors require a bit of extra memory as overhead.

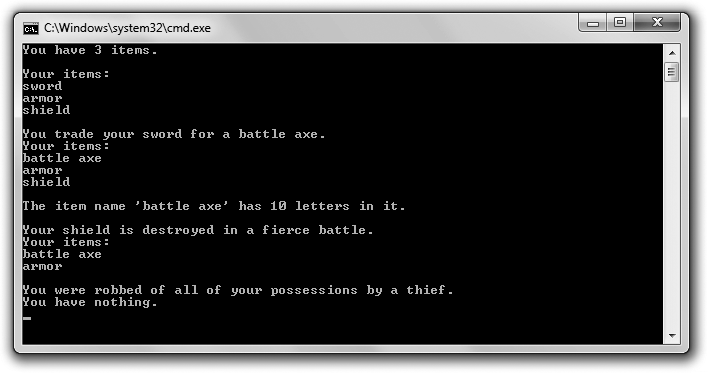
n There can be a performance cost when a vector grows in size.

n Vectors may not be available on some game console systems.

Overall, vectors (and the STL) can be a welcome tool in most any project.

#### [Introducing the Hero’s Inventory 2.0 Program](#_bookmark5)

From the user’s point of view, the Hero’s Inventory 2.0 program is similar to its predecessor, the Hero’s Inventory program from Chapter 3. The new version stores and works with a collection of string objects that represent a hero’s inventory. However, from the programmer’s perspective the program is quite different. That’s because the new program uses a vector instead of an array to represent the inventory. Figure 4.1 shows the results of the program.



**Figure 4.1**

This time the hero’s inventory is represented by a vector.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 4 folder; the filename is heros\_inventory2.cpp.

118 Chapter 4 n The Standard Template Library: Hangman

// Hero’s Inventory 2.0

// Demonstrates vectors

#include <iostream> #include <string> #include <vector>

using namespace std; int main()

{

vector<string> inventory; inventory.push\_back("sword"); inventory.push\_back("armor"); inventory.push\_back("shield");

cout << "You have " << inventory.size() << " items.\n"; cout << "\nYour items:\n";

for (unsigned int i = 0; i < inventory.size(); ++i)

{

cout << inventory[i] << endl;

}

cout << "\nYou trade your sword for a battle axe."; inventory[0] = "battle axe";

cout << "\nYour items:\n";

for (unsigned int i = 0; i < inventory.size(); ++i)

{

cout << inventory[i] << endl;

}

cout << "\nThe item name ’" << inventory[0] << "’ has "; cout << inventory[0].size() << " letters in it.\n";

cout << "\nYour shield is destroyed in a fierce battle."; inventory.pop\_back();

cout << "\nYour items:\n";

for (unsigned int i = 0; i < inventory.size(); ++i)

{

Using Vectors 119

cout << inventory[i] << endl;

}

cout << "\nYou were robbed of all of your possessions by a thief."; inventory.clear();

if (inventory.empty())

{

}

else

{

}

cout << "\nYou have nothing.\n";

cout << "\nYou have at least one item.\n";

return 0;

}

#### [Preparing to Use Vectors](#_bookmark5)

Before I can declare a vector, I have to include the file that contains its definition:

#include <vector>

All STL components live in the std namespace, so by using the following code (as I typically do) I can refer to a vector without having to precede it with std::.

using namespace std;

#### [Declaring a Vector](#_bookmark5)

Okay, the first thing I do in main() is declare a new vector.

vector<string> inventory;

The preceding line declared an empty vector named inventory, which can contain string object elements. Declaring an empty vector is fine because it grows in size when you add new elements.

To declare a vector of your own, write vector followed by the type of objects you want to use with the vector (surrounded by the < and > symbols), followed by the vector name.

120 Chapter 4 n The Standard Template Library: Hangman

**Hi n t**

There are additional ways to declare a vector. You can declare one with a starting size by specifying a number in parentheses after the vector name.

vector<string> inventory(10);

The preceding code declared a vector to hold string object elements with a starting size of 10. You can also initialize all of a vector’s elements to the same value when you declare it. You simply supply the number of elements followed by the starting value, as in:

vector<string> inventory(10, "nothing");

The preceding code declared a vector with a size of 10 and initialized all 10 elements to

"nothing". Finally, you can declare a vector and initialize it with the contents of another vector.

vector<string> inventory(myStuff);

The preceding code created a new vector with the same contents as the vector myStuff.

#### [Using the push\_back() Member Function](#_bookmark5)

Next I give the hero the same three starting items as in the previous version of the program.

inventory.push\_back("sword"); inventory.push\_back("armor"); inventory.push\_back("shield");

The push\_back() member function adds a new element to the end of a vector. In the preceding lines I added "sword", "armor", and "shield" to inventory. As a result, inventory[0] is equal to "sword", inventory[1] is equal to "armor", and inventory[2] is equal to "shield".

#### [Using the size() Member Function](#_bookmark5)

Next I display the number of items the hero has in his possession.

cout << "You have " << inventory.size() << " items.\n";

I get the size of inventory by calling the size() member function with inventory.size(). The size() member function simply returns the size of a vector. In this case, it returns 3.

Using Vectors 121

#### [Indexing Vectors](#_bookmark5)

Next I display all of the hero’s items.

cout << "\nYour items:\n";

for (unsigned int i = 0; i < inventory.size(); ++i)

{

cout << inventory[i] << endl;

}

Just as with arrays, you can index vectors by using the subscripting operator. In fact, the preceding code is nearly identical to the same section of code from the original Hero’s Inventory program. The only difference is that I used inventory. size() to specify when the loop should end. Note that I made the loop variable i an unsigned int because the value returned by size() is an unsigned integral type.

Next I replace the hero’s first item.

inventory[0] = "battle axe";

Again, just as with arrays, I use the subscripting operator to assign a new value to an existing element position.

**Trap**

Although vectors are dynamic, you can’t increase a vector’s size by applying the subscripting operator. For example, the following highly dangerous code snippet does not increase the size of the vector inventory:

vector<string> inventory; //creating an empty vector inventory[0] = "sword"; //may cause your program to crash!

Just as with arrays, you can attempt to access a nonexistent element position—but with potentially disastrous results. The preceding code changed some unknown section of your computer’s memory and could cause your program to crash. To add a new element at the end of a vector, use the push\_back() member function.

#### [Calling Member Functions of an Element](#_bookmark5)

Next I show the number of letters in the name of the first item in the hero’s inventory.

cout << inventory[0].size() << " letters in it.\n";

122 Chapter 4 n The Standard Template Library: Hangman

Just as with arrays, you can access the member functions of a vector element by writing the element, followed by the member selection operator, followed by the member function name. Because inventory[0] is equal to "battle axe", inventory[0].size() returns 10.

#### [Using the pop\_back() Member Function](#_bookmark5)

I remove the hero’s shield using

inventory.pop\_back();

The pop\_back() member function removes the last element of a vector and reduces the vector size by one. In this case, inventory.pop\_back() removes "shield" from inventory because that was the last element in the vector. Also, the size of inventory is reduced from 3 to 2.

#### [Using the clear() Member Function](#_bookmark5)

Next I simulate the act of a thief robbing the hero of all of his items.

inventory.clear();

The clear() member function removes all of the items of a vector and sets its size to 0. After the previous line of code executes, inventory is an empty vector.

#### [Using the empty() Member Function](#_bookmark5)

Finally, I check to see whether the hero has any items in his inventory.

if (inventory.empty())

{

cout << "\nYou have nothing.\n";

}

else

{

cout << "\nYou have at least one item.\n";

}

The vector member function empty() works just like the string member function empty(). It returns true if the vector object is empty; otherwise, it returns false. Because inventory is empty in this case, the program displays the message, “You have nothing.”

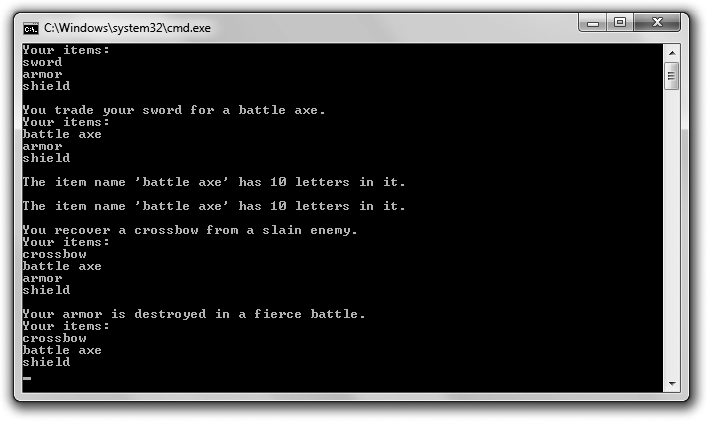
Using Iterators 123

### [Using Iterators](#_bookmark5)

Iterators are the key to using containers to their fullest potential. With iterators you can, well, iterate through a sequence container. In addition, important parts of the STL require iterators. Many container member functions and STL algorithms take iterators as arguments. So if you want to reap the benefits of these member functions and algorithms, you’ve got to use iterators.

#### [Introducing the Hero’s Inventory 3.0 Program](#_bookmark5)

The Hero’s Inventory 3.0 program acts like its two predecessors, at least at the start. The program shows off a list of items, replaces the first item, and displays the number of letters in the name of an item. But then the program does something new: It inserts an item at the beginning of the group, and then it removes an item from the middle of the group. The program accomplishes all of this by working with iterators. Figure 4.2 shows the program in action.



**Figure 4.2**

The program performs a few vector manipulations that you can accomplish only with iterators.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 4 folder; the filename is heros\_inventory3.cpp.

124 Chapter 4 n The Standard Template Library: Hangman

// Hero’s Inventory 3.0

// Demonstrates iterators

#include <iostream> #include <string> #include <vector>

using namespace std; int main()

{

vector<string> inventory; inventory.push\_back("sword"); inventory.push\_back("armor"); inventory.push\_back("shield");

vector<string>::iterator myIterator; vector<string>::const\_iterator iter;

cout << "Your items:\n";

for (iter = inventory.begin(); iter != inventory.end(); ++iter)

{

cout << \*iter << endl;

}

cout << "\nYou trade your sword for a battle axe."; myIterator = inventory.begin();

\*myIterator = "battle axe"; cout << "\nYour items:\n";

for (iter = inventory.begin(); iter != inventory.end(); ++iter)

{

cout << \*iter << endl;

}

cout << "\nThe item name ’" << \*myIterator << "’ has "; cout << (\*myIterator).size() << " letters in it.\n";

cout << "\nThe item name ’" << \*myIterator << "’ has "; cout << myIterator->size() << " letters in it.\n";

Using Iterators 125

cout << "\nYou recover a crossbow from a slain enemy."; inventory.insert(inventory.begin(), "crossbow"); cout << "\nYour items:\n";

for (iter = inventory.begin(); iter != inventory.end(); ++iter)

{

cout << \*iter << endl;

}

cout << "\nYour armor is destroyed in a fierce battle."; inventory.erase((inventory.begin() + 2));

cout << "\nYour items:\n";

for (iter = inventory.begin(); iter != inventory.end(); ++iter)

{

cout << \*iter << endl;

}

return 0;

}

#### [Declaring Iterators](#_bookmark5)

After I declare a vector for the hero’s inventory and add the same three string

objects from the previous incarnations of the program, I declare an iterator.

vector<string>::iterator myIterator;

The preceding line declares an iterator named myIterator for a vector that contains string objects. To declare an iterator of you own, follow the same pattern. Write the container type, followed by the type of objects the container will hold (surrounded by the < and > symbols), followed by the scope resolution operator (the :: symbol), followed by iterator, followed by a name for your new iterator.

So what are iterators? *Iterators* are values that identify a particular element in a container. Given an iterator, you can access the value of the element. Given the right kind of iterator, you can change the value. Iterators can also move among elements via familiar arithmetic operators.

A way to think about iterators is to imagine them as Post-it notes that you can stick on a specific element in a container. An iterator is not one of the elements, but a way to refer to one. Specifically, I can use myIterator to refer to a particular element of the vector inventory. That is, I can stick the myIterator

126 Chapter 4 n The Standard Template Library: Hangman

Post-it note on a specific element in inventory. Once I’ve done that, I can access the element or even change it through the iterator.

Next, I declare another iterator.

vector<string>::const\_iterator iter;

The preceding line of code creates a constant iterator named iter for a vector that contains string objects. A *constant iterator* is just like a regular iterator except that you can’t use it to change the element to which it refers; the element must remain constant. You can think of a constant iterator as providing read- only access. However, the iterator itself can change. This means you can move iter all around the vector inventory as you see fit. You can’t, however, change the value of any of the elements through iter. With a constant iterator the Post- it can change, but the thing it’s stuck to can’t.

Why would you want to use a constant iterator if it’s a limited version of a regular iterator? First, it makes your intentions clearer. When you use a constant iterator, it’s clear that you won’t be changing any element to which it refers. Second, it’s safer. You can use a constant iterator to avoid accidentally changing a container element. (If you attempt to change an element through a constant iterator, you’ll generate a compile error.)

**Trap**

Using push\_back() might invalidate all iterators referencing the vector.

Is all of this iterator talk a little too abstract for you? Are you tired of analogies about Post-it notes? Fear not—next, I put an actual iterator to work.

#### [Looping through a Vector](#_bookmark5)

Next, I loop through the contents of the vector and display the hero’s inventory.

cout << "Your items:\n";

for (iter = inventory.begin(); iter != inventory.end(); ++iter)

{

cout << \*iter << endl;

}

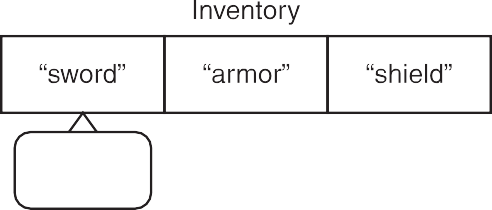
In the preceding code, I use a for loop to move from the first to the last element of inventory. At this general level, this is exactly how I looped through the

Using Iterators 127

contents of the vector in Hero’s Inventory 2.0. But instead of using an integer and the subscripting operator to access each element, I used an iterator. Basically, I moved the Post-it note through the entire sequence of elements and displayed the value of each element to which the note was stuck. There are a lot of new ideas in this little loop, so I’ll tackle them one at a time.

***Calling the begin() Vector Member Function***

In the initialization statement of the loop, I assign the return value of inventory. begin() to iter. The begin() member function returns an iterator that refers to a container’s first element. So in this case, the statement assigns an iterator that refers to the first element of inventory (the string object equal to "sword") to iter. Figure 4.3 shows an abstract view of the iterator returned by a call to inventory.begin(). (Note that the figure is abstract because the vector inventory doesn’t contain the string literals "sword", "armor", and "shield"; it contains string objects.)





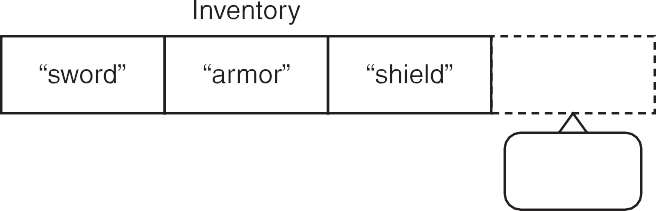
**Figure 4.3**

A call to inventory.begin() returns an iterator that refers to the first element in the vector.

***Calling the end() Vector Member Function***

In the test statement of the loop, I test the return value of inventory.end() against iter to make sure the two are not equal. The end() member function returns an iterator one past the last element in a container. This means the loop will continue until iter has moved through all of the elements in inventory. Figure 4.4 shows an abstract view of the iterator returned by a call to this member function. (Note that the figure is abstract because the vector inventory doesn’t contain the string literals "sword", "armor", and "shield"; it contains string objects.)

128 Chapter 4 n The Standard Template Library: Hangman



**Figure 4.4**

A call to inventory.end() returns an iterator one past the last element of the vector.

**Trap**

The end() vector member function returns an iterator that’s one *past* the last element in the vector—not the last element. Therefore, you can’t get a value from the iterator returned by end(). This might seem counter-intuitive, but it works well for loops that move through a container.

***Altering an Iterator***

The action statement in the loop, ++iter, increments iter, which moves it to the next element in the vector. Depending upon the iterator, you can perform other mathematical operations on iterators to move them around a container. Most often, though, you’ll find that you simply want to increment an iterator.

***Dereferencing an Iterator***

In the loop body, I send \*iter to cout. By placing the dereference operator (\*) in front of iter, I display the value of the element to which the iterator refers (not the iterator itself). By placing the dereference operator in front of an iterator, you’re saying, “Treat this as the thing that the iterator references, not as the iterator itself.”

#### [Changing the Value of a Vector Element](#_bookmark5)

Next, I change the first element in the vector from the string object equal to "sword" to the string object equal to "battle axe". First I set myIterator to reference the first element of inventory.

myIterator = inventory.begin();

Then I change the value of the first element.

Using Iterators 129

\*myIterator = "battle axe";

Remember, by dereferencing myIterator with \*, the preceding assignment statement says, “Assign "battle axe" to the element that myIterator references.” It does not change myIterator. After the assignment statement, myIterator still refers to the first element in the vector.

Just to prove that the assignment worked, I then display all of the elements in

inventory.

#### [Accessing Member Functions of a Vector Element](#_bookmark5)

Next I display the number of characters in the name of the first item in the hero’s inventory.

cout << "\nThe item name ’" << \*myIterator << "’ has "; cout << (\*myIterator).size() << " letters in it.\n";

The code (\*myIterator).size() says, “Take the result of dereferencing myIterator and call that object’s size() member function.” Because myIterator refers to the string object equal to "battle axe", the code returns 10.

**Hint**

Whenever you dereference an iterator to access a data member or member function, surround the dereferenced iterator by a pair of parentheses. This ensures that the dot operator will be applied to the object the iterator references.

The code (\*myIterator).size() is not the prettiest, so C++ offers an alternative, more intuitive way to express the same thing, which I demonstrate in the next two lines of the program.

cout << "\nThe item name ’" << \*myIterator << "’ has "; cout << myIterator->size() << " letters in it.\n";

The preceding code does exactly the same thing the first pair of lines I presented in this section do; it displays the number of characters in "battle axe". However, notice that I substitute myIterator->size() for (\*myIterator).size(). You can see that this version (with the -> symbol) is more readable. The two pieces of code mean exactly the same thing to the computer, but this new version is easier for humans to use. In general, you can use the -> operator to access the member functions or data members of an object that an iterator references.

130 Chapter 4 n The Standard Template Library: Hangman

**Hi n t**

*Syntactic sugar* is a nicer, alternative syntax. It replaces harsh syntax with something that’s a bit easier to swallow. As an example, instead of writing the code (\*myIterator).size(), I can use the syntactic sugar provided by the -> operator and write myIterator->size().

#### [Using the insert() Vector Member Function](#_bookmark5)

Next I add a new item to the hero’s inventory. This time, though, I don’t add the item to the end of the sequence; instead, I insert it at the beginning.

inventory.insert(inventory.begin(), "crossbow");

One form of the insert() member function inserts a new element into a vector just before the element referred to by a given iterator. You supply two arguments to this version of insert()—the first is an iterator, and the second is the element to be inserted. In this case, I inserted "crossbow" into inventory just before the first element. As a result, all of the other elements will move down by one. This version of the insert() member function returns an iterator that references the newly inserted element. In this case, I don’t assign the returned iterator to a variable.

**Trap**

Calling the insert() member function on a vector invalidates all of the iterators that reference elements after the insertion point because all of the elements after the insertion point are shifted down by one.

Next I show the contents of the vector to prove the insertion worked.

#### [Using the erase() Vector Member Function](#_bookmark5)

Next I remove an item from the hero’s inventory. However, this time I don’t remove the item at the end of the sequence; instead, I remove one from the middle.

inventory.erase((inventory.begin() + 2));

One form of the erase() member function removes an element from a vector. You supply one argument to this version of erase()—the iterator that references the element you want to remove. In this case, I passed (inventory.begin() + 2), which is equal to the iterator that references the third element in inventory. This removes the string object equal to "armor". As a result, all of the following elements will move up by one. This version of the erase() member function

Using Algorithms 131

returns an iterator that references the element after the element that was removed. In this case, I don’t assign the returned iterator to a variable.

**Trap**

Calling the erase() member function on a vector invalidates all of the iterators that reference elements after the removal point because all of the elements after the removal point are shifted up by one.

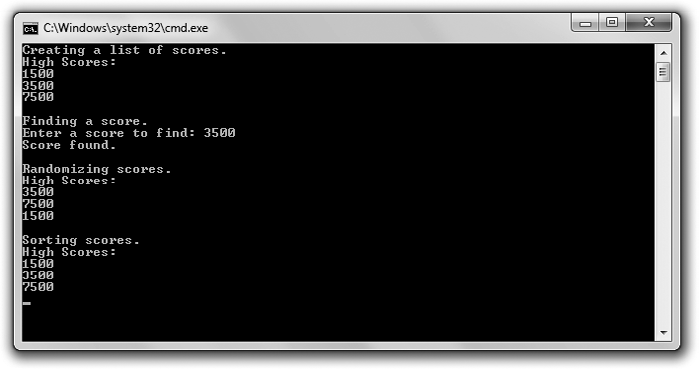
Next I show the contents of the vector to prove the removal worked.

### [Using Algorithms](#_bookmark5)

The STL defines a group of algorithms that allow you to manipulate elements in containers through iterators. Algorithms exist for common tasks such as searching, randomizing, and sorting. These algorithms are your built-in arsenal of flexible and efficient weapons. By using them, you can leave the mundane task of manipulating container elements in common ways to the STL so you can concentrate on writing your game. The powerful thing about these algorithms is that they are generic—the same algorithm can work with elements of different container types.

#### [Introducing the High Scores Program](#_bookmark5)

The High Scores program creates a vector of high scores. It uses STL algorithms to search, shuffle, and sort the scores. Figure 4.5 illustrates the program.



**Figure 4.5**

STL algorithms search, shuffle, and sort elements of a vector of high scores.

132 Chapter 4 n The Standard Template Library: Hangman

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 4 folder; the filename is high\_scores.cpp.

// High Scores

// Demonstrates algorithms

#include <iostream> #include <vector> #include <algorithm> #include <ctime> #include <cstdlib>

using namespace std; int main()

{

vector<int>::const\_iterator iter;

cout << "Creating a list of scores."; vector<int> scores; scores.push\_back(1500); scores.push\_back(3500); scores.push\_back(7500);

cout << "\nHigh Scores:\n";

for (iter = scores.begin(); iter != scores.end(); ++iter)

{

cout << \*iter << endl;

}

cout << "\nFinding a score."; int score;

cout << "\nEnter a score to find: "; cin >> score;

iter = find(scores.begin(), scores.end(), score); if (iter != scores.end())

Using Algorithms 133

{

}

else

{

}

cout << "Score found.\n";

cout << "Score not found.\n";

cout << "\nRandomizing scores."; srand(static\_cast<unsigned int>(time(0))); random\_shuffle(scores.begin(), scores.end()); cout << "\nHigh Scores:\n";

for (iter = scores.begin(); iter != scores.end(); ++iter)

{

cout << \*iter << endl;

}

cout << "\nSorting scores."; sort(scores.begin(), scores.end()); cout << "\nHigh Scores:\n";

for (iter = scores.begin(); iter != scores.end(); ++iter)

{

cout << \*iter << endl;

}

return 0;

}

#### [Preparing to Use Algorithms](#_bookmark5)

So that I can use the STL algorithms, I include the file with their definitions.

#include <algorithm>

As you know, all STL components live in the std namespace. By using the following code (as I typically do), I can refer to algorithms without having to precede them with std::.

using namespace std;

134 Chapter 4 n The Standard Template Library: Hangman

#### [Using the find() Algorithm](#_bookmark5)

After I display the contents of the vector scores, I get a value from the user to find and store it in the variable score. Then I use the find() algorithm to search the vector for the value:

iter = find(scores.begin(), scores.end(), score);

The find() STL algorithm searches a specified range of a container’s elements for a value. It returns an iterator that references the first matching element. If no match is found, it returns an iterator to the end of the range. You must pass the starting point as an iterator, the ending point as an iterator, and a value to find. The algorithm searches from the starting iterator up to but not including the ending iterator. In this case, I passed scores.begin() and scores.end() as the first and second arguments to search the entire vector. I passed score as the third argument to search for the value the user entered.

Next I check to see if the value score was found:

if (iter != scores.end())

{

cout << "Score found.\n";

}

else

{

cout << "Score not found.\n";

}

Remember, iter will reference the first occurrence of score in the vector, if the value was found. So, as long as iter is not equal to scores.end(), I know that score was found and I display a message saying so. Otherwise, iter will be equal to scores.end() and I know score was not found.

#### [Using the random\_shuffle() Algorithm](#_bookmark5)

Next I prepare to randomize the scores using the random\_shuffle() algorithm. Just as when I generate a single random number, I seed the random number generator before I call random\_shuffle(), so the order of the scores might be different each time I run the program.

srand(static\_cast<unsigned int>(time(0)));

Then I reorder the scores in a random way.

random\_shuffle(scores.begin(), scores.end());

Using Algorithms 135

The random\_shuffle() algorithm randomizes the elements of a sequence. You must supply as iterators the starting and ending points of the sequence to shuffle. In this case, I passed the iterators returned by scores.begin() and scores.end(). These two iterators indicate that I want to shuffle all of the elements in scores. As a result, scores contains the same scores, but in some random order.

Then I display the scores to prove the randomization worked.

**Trick**

Although you might not want to randomize a list of high scores, random\_shuffle() is a valuable algorithm for games. You can use it for everything from shuffling a deck of cards to mixing up the order of the enemies a player will encounter in a game level.

#### [Using the sort() Algorithm](#_bookmark5)

Next I sort the scores.

sort(scores.begin(), scores.end());

The sort() algorithm sorts the elements of a sequence in ascending order. You must supply as iterators the starting and ending points of the sequence to sort. In this particular case, I passed the iterators returned by scores.begin() and scores.end(). These two iterators indicate that I want to sort all of the elements in scores. As a result, scores contains all of the scores in ascending order.

Finally, I display the scores to prove the sorting worked.

**Trick**

A very cool property of STL algorithms is that they can work with containers defined outside of the STL. These containers only have to meet certain requirements. For example, even though string objects are not part of the STL, you can use appropriate STL algorithms on them. The following code snippet demonstrates this:

string word = "High Scores"; random\_shuffle(word.begin(), word.end());

The preceding code randomly shuffles the characters in word. As you can see, string objects have both begin() and end() member functions, which return iterators to the first character and one past the last character, respectively. That’s part of the reason why STL algorithms work with strings—because they’re designed to.

136 Chapter 4 n The Standard Template Library: Hangman

### [Understanding Vector Performance](#_bookmark6)

Like all STL containers, vectors provide game programmers with sophisticated ways to work with information, but this level of sophistication can come at a performance cost. And if there’s one thing game programmers obsess about, it’s performance. But fear not, vectors and other STL containers are incredibly efficient. In fact, they’ve already been used in published PC and console games. However, these containers have their strengths and weaknesses; a game programmer needs to understand the performance characteristics of the various container types so that he can choose the right one for the job.

#### [Examining Vector Growth](#_bookmark6)

Although vectors grow dynamically as needed, every vector has a specific size. When a new element added to a vector pushes the vector beyond its current size, the computer reallocates memory and might even copy all of the vector elements to this newly seized chunk of memory real estate. This can cause a performance hit.

The most important thing to keep in mind about program performance is whether or not you need to care. For example, vector memory reallocation might not occur at a performance-critical part of your program. In that case, you can safely ignore the cost of reallocation. Also, with small vectors, the reallocation cost might be insignificant so, again, you can safely ignore it. However, if you need greater control over when these memory reallocations occur, you have it.

***Using the capacity() Member Function***

The capacity() vector member function returns the capacity of a vector—in other words, the number of elements that a vector can hold before a program must reallocate more memory for it. A vector’s capacity is not the same thing as its size (the number of elements a vector currently holds). Here’s a code snippet to help drive this point home:

cout << "Creating a 10 element vector to hold scores.\n"; vector<int> scores(10, 0); //initialize all 10 elements to 0 cout << "Vector size is :" << scores.size() << endl;

cout << "Vector capacity is:" << scores.capacity() << endl;

Understanding Vector Performance 137

cout << "Adding a score.\n";

scores.push\_back(0); //memory is reallocated to accommodate growth cout << "Vector size is :" << scores.size() << endl;

cout << "Vector capacity is:" << scores.capacity() << endl;

Right after I declare and initialize the vector, this code reports that its size and capacity are both 10. However, after an element is added, the code reports that the vector’s size is 11 while its capacity is 20. That’s because the capacity of a vector doubles every time a program reallocates additional memory for it. In this case, when a new score was added, memory was reallocated, and the capacity of the vector doubled from 10 to 20.

***Using the reserve() Member Function***

The reserve() member function increases the capacity of a vector to the number supplied as an argument. Using reserve() gives you control over when a reallocation of additional memory occurs. Here’s an example:

cout << "Creating a list of scores.\n";

vector<int> scores(10, 0); //initialize all 10 elements to 0 cout << "Vector size is :" << scores.size() << endl;

cout << "Vector capacity is:" << scores.capacity() << endl;

cout << "Reserving more memory.\n";

scores.reserve(20); //reserve memory for 10 additional elements cout << "Vector size is :" << scores.size() << endl;

cout << "Vector capacity is:" << scores.capacity() << endl;

Right after I declare and initialize the vector, this code reports that its size and capacity are both 10. However, after I reserve memory for 10 additional elements, the code reports that the vector’s size is still 10 while its capacity is 20.

By using reserve() to keep a vector’s capacity large enough for your purposes, you can delay memory reallocation to a time of your choosing.

**Hint**

As a beginning game programmer, it’s good to be aware of how vector memory allocation works; however, don’t obsess over it. The first game programs you’ll write probably won’t benefit from a more manual process of vector memory allocation.

138 Chapter 4 n The Standard Template Library: Hangman

#### [Examining Element Insertion and Deletion](#_bookmark6)

Adding or removing an element from the end of a vector using the push\_back() or pop\_back() member functions is extremely efficient. However, adding or removing an element at any other point in a vector (for example, using insert() or erase()) can require more work because you might have to move multiple elements to accommodate the insertion or deletion. With small vectors the overhead is usually insignificant, but with larger vectors (with, say, thousands of elements), inserting or erasing elements from the middle of a vector can cause a performance hit.

Fortunately, the STL offers another sequence container type, list, which allows for efficient insertion and deletion regardless of the sequence size. The important thing to remember is that one container type isn’t the solution for every problem. Although vector is versatile and perhaps the most popular STL container type, there are times when another container type might make more sense.

**Trap**

Just because you want to insert or delete elements from the middle of a sequence, that doesn’t mean you should abandon the vector. It might still be a good choice for your game program. It really depends on how you use the sequence. If your sequence is small or there are only a few insertion and deletions, then a vector might still be your best bet.

### [Examining Other STL Containers](#_bookmark6)

The STL defines a variety of container types that fall into two basic categories— sequential and associative. With a *sequential container*, you can retrieve values in sequence, while an *associative container* lets you retrieve values based on keys. vector is an example of a sequential container.

How might you use these different container types? Consider an online, turned- based strategy game. You could use a sequential container to store a group of players that you want to cycle through in, well, sequence. On the other hand, you could use an associative container to retrieve player information in a random- access fashion by looking up a unique identifier, such as a player’s IP address.

Finally, the STL defines container adaptors that adapt one of the sequence containers. *Container adaptors* represent standard computer science data

Planning Your Programs 139

structures. Although they are not official containers, they look and feel just like them. Table 4.1 lists the container types offered by the STL.

Double-ended queue

Linear list

Collection of key/value pairs in which each key is associated with exactly one value

Collection of key/value pairs in which each key may be associated with more than one value

Collection in which each element is not necessarily unique Priority queue

Queue

Collection in which each element is unique Stack

Dynamic array

multiset Associative

priority\_queue Adaptor

queue Adaptor

set Associative

stack Adaptor

vector Sequential

Associative

multimap

Sequential

Sequential Associative

deque

list map

**Description**

**Table 4.1** STL Containers

**Container Type**

### [Planning Your Programs](#_bookmark6)

So far, all the programs you’ve seen have been pretty simple. The idea of formally planning any of them on paper probably seems like overkill. It’s not— planning your programs (even the small ones) will almost always result in time (and frustration) saved.

Programming is a lot like construction. Imagine a contractor building a house for you without a blueprint. Yikes! You might end up with a house that has 12 bathrooms, no windows, and a front door on the second floor. Plus, it probably would cost you 10 times the estimated price. Programming is the same way. Without a plan, you’ll likely struggle through the process and waste time. You might even end up with a program that doesn’t quite work.

#### [Using Pseudocode](#_bookmark6)

Many programmers sketch out their programs using *pseudocode*—a language that falls somewhere between English and a formal programming language. Anyone who understands English should be able to follow pseudocode. Here’s

140 Chapter 4 n The Standard Template Library: Hangman

an example: Suppose I want to make a million dollars. A worthy goal, but what do I do to achieve it? I need a plan. So I come up with one and put it in pseudocode.

*If you can think of a new and useful product Then that’s your product*

*Otherwise*

*Repackage an existing product as your product Make an infomercial about your product*

*Show the infomercial on TV*

*Charge $100 per unit of your product Sell 10,000 units of your product*

Even though anyone, even a non-programmer, can understand my plan, my pseudocode feels vaguely like a program. The first four lines resemble an if statement with an else clause, and that’s intentional. When you write your plan, you should try to incorporate the feel of the code that you’re representing with pseudocode.

#### [Using Stepwise Refinement](#_bookmark6)

Your programming plan might not be finished after only one draft. Often, pseudocode needs multiple passes before it can be implemented in programming code. *Stepwise refinement* is one process used to rewrite pseudocode to make it ready for implementation. Stepwise refinement is pretty simple. Basically, it means, “Make it more detailed.” By taking each step described in pseudocode and breaking it down into a series of simpler steps, the plan becomes closer to programming code. Using stepwise refinement, you keep breaking down each step until you feel the entire plan could be fairly easily translated into a program. As an example, take a step from my master plan to make a million dollars:

*Create an infomercial about your product*

This might seem like too vague of a task. How do you create an infomercial? Using stepwise refinement, you can break down the single step into several others. So it becomes

*Write a script for an infomercial about your product Rent a TV studio for a day*

*Hire a production crew*

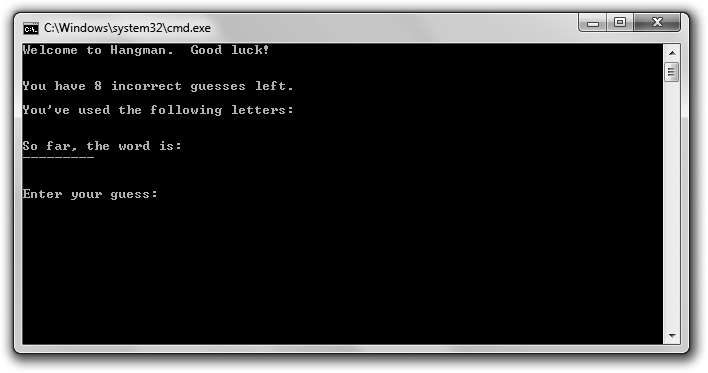
*Hire an enthusiastic audience Film the infomercial*

Introducing Hangman 141

If you feel these five steps are clear and achievable, then that part of the pseudocode has been thoroughly refined. If you’re still unclear about a step, refine it some more. Continue with this process and you will have a complete plan—and a million dollars.

### [Introducing Hangman](#_bookmark6)

In the Hangman program, the computer picks a secret word and the player tries to guess it one letter at a time. The player is allowed eight incorrect guesses. If he or she fails to guess the word in time, the player is hanged and the game is over. Figure 4.6 shows the game.



**Figure 4.6**

The Hangman game in action

#### [Planning the Game](#_bookmark6)

Before I write a single line in C++, I plan the game program using pseudocode.

*Create a group of words*

*Pick a random word from the group as the secret word*

*While player hasn’t made too many incorrect guesses and hasn’t guessed the secret word*

*Tell player how many incorrect guesses he or she has left Show player the letters he or she has guessed*

142 Chapter 4 n The Standard Template Library: Hangman

*Show player how much of the secret word he or she has guessed Get player’s next guess*

*While player has entered a letter that he or she has already guessed Get player’s guess*

*Add the new guess to the group of used letters If the guess is in the secret word*

*Tell the player the guess is correct*

*Update the word guessed so far with the new letter Otherwise*

*Tell the player the guess is incorrect*

*Increment the number of incorrect guesses the player has made If the player has made too many incorrect guesses*

*Tell the player that he or she has been hanged Otherwise*

*Congratulate the player on guessing the secret word*

Although the pseudocode doesn’t account for every line of C++ I’ll write, I think it does a good job describing what I need to do. Then I begin writing the program.

#### [Setting Up the Program](#_bookmark6)

As usual, I start with some comments and include the files I need.

// Hangman

// The classic game of hangman

#include <iostream> #include <string> #include <vector> #include <algorithm> #include <ctime> #include <cctype>

using namespace std;

Notice that I include a new file—cctype. It’s part of the standard library and it includes functions for converting characters to uppercase, which I use so I can compare apples to apples (uppercase to uppercase) when I compare individual characters.

Introducing Hangman 143

#### [Initializing Variables and Constants](#_bookmark6)

Next I start the main() function and initialize variables and constants for the game.

int main()

{

//setup

const int MAX\_WRONG = 8; //maximum number of incorrect guesses allowed

vector<string> words; //collection of possible words to guess words.push\_back("GUESS");

words.push\_back("HANGMAN"); words.push\_back("DIFFICULT");

srand(static\_cast<unsigned int>(time(0))); random\_shuffle(words.begin(), words.end());

const string THE\_WORD = words[0]; //word to guess

int wrong = 0; //number of incorrect guesses string soFar(THE\_WORD.size(), ’-’); //word guessed so far

string used = ""; //letters already guessed

cout << "Welcome to Hangman. Good luck!\n";

MAX\_WRONG is the maximum number of incorrect guesses the player can make. words is a vector of possible words to guess. I randomize words using the random\_shuffle() algorithm, and then I assign the first word in the vector to THE\_WORD, which is the secret word the player must guess. wrong is the number of incorrect guesses the player has made. soFar is the word guessed so far by the player. soFar starts out as a series of dashes—one for each letter in the secret word. When the player guesses a letter that’s in the secret word, I replace the dash at the corresponding position with the letter.

#### [Entering the Main Loop](#_bookmark6)

Next I enter the main loop, which continues until the player has made too many incorrect guesses or has guessed the word.

//main loop

while ((wrong < MAX\_WRONG) && (soFar != THE\_WORD))

{

144 Chapter 4 n The Standard Template Library: Hangman

cout << "\n\nYou have " << (MAX\_WRONG - wrong); cout << " incorrect guesses left.\n";

cout << "\nYou’ve used the following letters:\n" << used << endl; cout << "\nSo far, the word is:\n" << soFar << endl;

#### [Getting the Player’s Guess](#_bookmark6)

Next I get the player’s guess.

char guess;

cout << "\n\nEnter your guess: "; cin >> guess;

guess = toupper(guess); //make uppercase since secret word in uppercase while (used.find(guess) != string::npos)

{

cout << "\nYou’ve already guessed " << guess << endl; cout << "Enter your guess: ";

cin >> guess;

guess = toupper(guess);

}

used += guess;

if (THE\_WORD.find(guess) != string::npos)

{

cout << "That’s right! " << guess << " is in the word.\n";

//update soFar to include newly guessed letter for (int i = 0; i < THE\_WORD.length(); ++i)

{

if (THE\_WORD[i] == guess)

{

soFar[i] = guess;

}

}

}

else

{

cout << "Sorry, " << guess << " isn’t in the word.\n";

++wrong;

}

}

Summary 145

I convert the guess to uppercase using the function uppercase(), which is defined in the file cctype. I do this so I can compare uppercase letters to uppercase letters when I’m checking a guess against the letters of the secret word.

If the player guesses a letter that he or she has already guessed, I make the player guess again. If the player guesses a letter correctly, I update the word guessed so far. Otherwise, I tell the player the guess is not in the secret word and I increase the number of incorrect guesses the player has made.

#### [Ending the Game](#_bookmark6)

At this point, the player has guessed the word or has made one too many incorrect guesses. Either way, the game is over.

//shut down

if (wrong == MAX\_WRONG)

{

}

else

{

}

cout << "\nYou’ve been hanged!";

cout << "\nYou guessed it!";

cout << "\nThe word was " << THE\_WORD << endl;

return 0;

}

I congratulate the player or break the bad news that he or she has been hanged. Then I reveal the secret word.

### [Summary](#_bookmark6)

In this chapter, you learned the following concepts:

n The Standard Template Library (STL) is a powerful collection of programming code that provides containers, algorithms, and iterators.

n Containers are objects that let you store and access collections of values of the same type.

146 Chapter 4 n The Standard Template Library: Hangman

n Algorithms defined in the STL can be used with its containers and provide common functions for working with groups of objects.

n Iterators are objects that identify elements in containers and can be manipulated to move among elements.

n Iterators are the key to using containers to their fullest. Many of the container member functions require iterators, and the STL algorithms require them too.

n To get the value referenced by an iterator, you must dereference the iterator using the dereference operator (\*).

n A vector is one kind of sequential container provided by the STL. It acts like a dynamic array.

n It’s very efficient to iterate through a vector. It’s also very efficient to insert or remove an element from the end of a vector.

n It can be inefficient to insert or delete elements from the middle of a vector, especially if the vector is large.

n Pseudocode, which falls somewhere between English and a programming language, is used to plan programs.

n Stepwise refinement is a process used to rewrite pseudocode to make it ready for implementation.

### [Questions and Answers](#_bookmark6)

Q: Why is the STL important?

A: Because it saves game programmers time and effort. The STL provides commonly used container types and algorithms.

Q: Is the STL fast?

A: Definitely. The STL has been honed by hundreds of programmers to eke out as much performance as possible on each supported platform.

Q: When should I use a vector instead of an array?

A: Almost always. Vectors are efficient and flexible. They do require a little more memory than arrays, but this tradeoff is almost always worth the benefits.

Questions and Answers 147

Q: Is a vector as fast as an array?

A: Accessing a vector element can be just as fast as accessing an array element. Also, iterating though a vector can be just as fast as iterating through an array.

Q: If I can use the subscripting operator with vectors, why would I ever need iterators?

A: There are several reasons. First, many of the vector member functions require iterators. (insert() and erase() are two examples.) Second, STL algorithms require iterators. And third, you can’t use the subscripting operator with most of the STL containers, so you’re going to have to learn to use iterators sooner or later.

Q: Which is the best way to access elements of a vector—through iterators or through the subscripting operator?

A: It depends. If you need random-element access, then the subscripting operator is a natural fit. If you need to use STL algorithms, then you must use iterators.

Q: What about iterating through the elements of a vector? Should I use the subscripting operator or an iterator?

A: You can use either method. However, an advantage of using an iterator is that it gives you the flexibility to substitute a different STL container in place of a vector (such as a list) without much code changing.

Q: Why does the STL define more than one sequential container type?

A: Different sequential container types have different performance properties.

They’re like tools in a toolbox; each tool is best suited for a different job.

Q: What are container adaptors?

A: Container adaptors are based on one of the STL sequence containers; they represent standard computer data structures. Although they are not official containers, they look and feel just like them.

Q: What’s a stack?

A: A data structure in which elements are removed in the reverse order from how they were added. This means that the last element added is the first one

148 Chapter 4 n The Standard Template Library: Hangman

removed. This is just like a real-life stack, from which you remove the last item you placed on the top of the stack.

Q: What’s a queue?

A: A data structure in which elements are removed in the same order they were added. This is just like a real-life queue, such as a line of people in which the first person in line gets served first.

Q: What’s a double-ended queue?

A: A queue in which elements can be added or removed from either end. Q: What’s a priority queue?

A: A data structure that supports finding and removing the element with the highest priority.

Q: When would I use pseudocode?

A: Any time you want to plan a program or section of code. Q: When would I use stepwise refinement?

A: When you want to get even more detailed with your pseudocode.

### [Discussion Questions](#_bookmark6)

1. Why should a game programmer use the STL?
2. What are the advantages of a vector over an array?
3. What types of game objects might you store with a vector?
4. How do performance characteristics of a container type affect the decision to use it?
5. Why is program planning important?

### [Exercises](#_bookmark6)

1. Write a program using vectors and iterators that allows a user to main- tain a list of his or her favorite games. The program should allow the user to list all game titles, add a game title, and remove a game title.

Exercises 149

1. Assuming that scores is a vector that holds elements of type int, what’s wrong with the following code snippet (meant to increment each element)?

vector<int>::iterator iter;

//increment each score

for (iter = scores.begin(); iter != scores.end(); ++iter)

{

iter++;

}

1. Write pseudocode for the Word Jumble game from Chapter 3.

*This page intentionally left blank*

|  |  |
| --- | --- |
|  |  |
| [chapter 5](#_bookmark6) | |

# [Functions: Mad Lib](#_bookmark6)

Every program you’ve seen so far has consisted of one function—main(). However, once your programs reach a certain size or level of complexity, it becomes hard to work with them like this. Fortunately, there are ways to break up big programs into smaller, bite-sized chunks of code. In this chapter, you’ll learn about one way—creating new functions. Specifically, you’ll learn to:

n Write new functions

n Accept values into your new functions through parameters

n Return information from your new functions through return values

n Work with global variables and constants

n Overload functions

n Inline functions

### [Creating Functions](#_bookmark6)

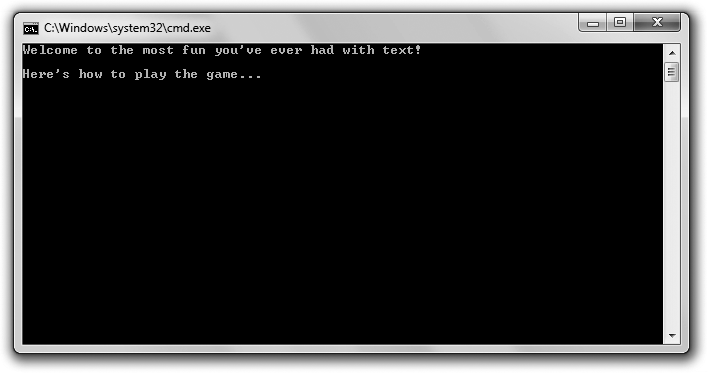
C++ lets you write programs with multiple functions. Your new functions work just like the ones that are part of the standard language—they go off and perform a task and then return control to your program. A big advantage of writing new functions is it allows you to break up your code into manageable pieces. Just like the functions you’ve already learned about from the standard library, your new functions should do one job well.

151

152 Chapter 5 n Functions: Mad Lib

#### [Introducing the Instructions Program](#_bookmark6)

The results of the Instructions program are pretty basic—a few lines of text that are the beginning of some game instructions. From the looks of the output, Instructions seems like a program you could have written way back in Chapter 1. But this program has a fresh element working behind the scenes—a new function. Take a look at Figure 5.1 to see the modest results of the code.



**Figure 5.1**

The instructions are displayed by a function.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 5 folder; the filename is instructions.cpp.

// Instructions

// Demonstrates writing new functions #include <iostream>

using namespace std;

// function prototype (declaration) void instructions();

Creating Functions 153

int main()

{

instructions(); return 0;

}

// function definition void instructions()

{

cout << "Welcome to the most fun you’ve ever had with text!\n\n"; cout << "Here’s how to play the game.. .\n";

}

#### [Declaring Functions](#_bookmark6)

Before you can call a function you’ve written, you have to declare it. One way to declare a function is to write a *function prototype*—code that describes the function. You write a prototype by listing the return value of the function (or void if the function returns no value), followed by the name of the function, followed by a list of parameters between a set of parentheses. *Parameters* receive the values sent as arguments in a function call.

Just before the main() function, I write a function prototype:

void instructions();

In the preceding code, I declared a function named instructions that doesn’t return a value. (You can tell this because I used void as the return type.) The function also takes no values so it has no parameters. (You can tell this because there’s nothing between the parentheses.)

Prototypes are not the only way to declare a function. Another way to accomplish the same thing is to let the function definition act as its own declaration. To do that, you simply have to put your function definition before the call to the function.

**Hint**

Although you don’t have to use prototypes, they offer a lot of benefits—not the least of which is making your code clearer.

154 Chapter 5 n Functions: Mad Lib

#### [Defining Functions](#_bookmark6)

Defining functions means writing all the code that makes the function tick. You define a function by listing the return value of the function (or void if the function returns no value), followed by the name of the function, followed by a list of parameters between a set of parentheses—just like a function prototype (except you don’t end the line with a semicolon). This is called the *function*

*header*. Then you create a block with curly braces that contains the instructions

to be executed when the function is executed. This is called the *function body*.

At the end of the Instructions program, I define my simple instructions() function, which displays some game instructions. Because the function doesn’t return any value, I don’t need to use a return statement like I do in main(). I simply end the function definition with a closing curly brace.

void instructions()

{

cout << "Welcome to the most fun you’ve ever had with text!\n\n"; cout << "Here’s how to play the game.. .\n";

}

**Trap**

A function definition must match its prototype on return type and function name; otherwise, you’ll generate a compile error.

#### [Calling Functions](#_bookmark6)

You call your own functions the same way you call any other function—by writing the function’s name followed by a pair of parentheses that encloses a valid list of arguments. In main(), I call my newly minted function simply with:

instructions();

This line invokes instructions(). Whenever you call a function, control of the program jumps to that function. In this case, it means control jumps to instructions() and the program executes the function’s code, which displays the game instructions. When a function finishes, control returns to the calling code. In this case, it means control returns to main(). The next statement in main () (return 0;) is executed and the program ends.

Using Parameters and Return Values 155

#### [Understanding Abstraction](#_bookmark6)

By writing and calling functions, you practice what’s known as *abstraction.* Abstraction lets you think about the big picture without worrying about the details. In this program, I can simply use the function instructions() without worrying about the details of displaying the text. All I have to do is call the function with one line of code, and it gets the job done.

You might be surprised where you find abstraction, but people use it all the time. For example, consider two employees at a fast-food restaurant. If one tells the other that he just filled a Number 3 and “sized it,” the other employee knows that the first employee took a customer’s order, went to the heat lamps, grabbed a burger, went over to the deep fryer, filled their biggest cardboard container with french fries, went to the soda fountain, grabbed their biggest cup, filled it with soda, gave it all to the customer, took the customer’s money, and gave the customer change. Not only would this level of detail make for a boring conversation, but it’s unnecessary. Both employees understand what it means to fill a Number 3 and “size it.” They don’t have to concern themselves with all the details because they’re using abstraction.

### [Using Parameters and Return Values](#_bookmark6)

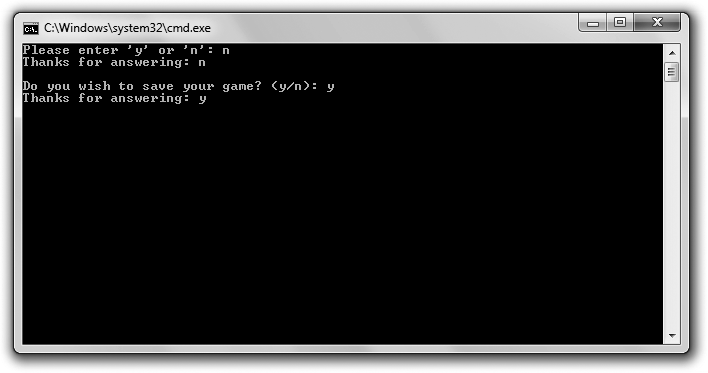
As you’ve seen with standard library functions, you can provide a function value and get a value back. For example, with the toupper() function, you provide a character, and the function returns the uppercase version of it. Your own functions can also receive values and return a value. This allows your functions to communicate with the rest of your program.

#### [Introducing the Yes or No Program](#_bookmark6)

The Yes or No program asks the user typical questions a gamer might have to answer. First, the program asks the user to indicate yes or no. Then the program gets more specific and asks whether the user wants to save his game. Again, the results of the program are not remarkable; it’s their implementation that’s interesting. Each question is posed by a different function that communicates with main(). Figure 5.2 shows a sample run of the program.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 5 folder; the filename is yes\_or\_no.cpp.

156 Chapter 5 n Functions: Mad Lib



**Figure 5.2**

Each question is asked by a separate function, and information is passed between these functions and

main().

// Yes or No

// Demonstrates return values and parameters

#include <iostream> #include <string>

using namespace std; char askYesNo1();

char askYesNo2(string question);

int main()

{

char answer1 = askYesNo1();

cout << "Thanks for answering: " << answer1 << "\n\n";

char answer2 = askYesNo2("Do you wish to save your game?"); cout << "Thanks for answering: " << answer2 << "\n";

return 0;

}

Using Parameters and Return Values 157

char askYesNo1()

{

char response1; do

{

cout << "Please enter ’y’ or ’n’: "; cin >> response1;

} while (response1 != ’y’ && response1 != ’n’);

return response1;

}

char askYesNo2(string question)

{

char response2; do

{

cout << question << " (y/n): "; cin >> response2;

} while (response2 != ’y’ && response2 != ’n’);

return response2;

}

#### [Returning a Value](#_bookmark6)

You can return a value from a function to send information back to the calling code. To return a value, you need to specify a return type and then return a value of that type from the function.

***Specifying a Return Type***

The first function I declare, askYesNo1(), returns a char value. You can tell this from the function prototype before main():

char askYesNo1();

You can also see this from the function definition after main():

char askYesNo1()

158 Chapter 5 n Functions: Mad Lib

***Using the return Statement***

askYesNo1() asks the user to enter y or n and keeps asking until he does. Once the user enters a valid character, the function wraps up with the following line, which returns the value of response1.

return response1;

Notice that response1 is a char value. It has to be because that’s what I promised to return in both the function prototype and function definition.

A function ends whenever it hits a return statement. It’s perfectly acceptable for a function to have more than one return. This just means that the function has several points at which it can end.

**Trick**

You don’t have to return a value with a return statement. You can use return by itself in a function that returns no value (one that indicates void as its return type) to end the function.

***Using a Returned Value***

In main(), I call the function with the following line, which assigns the return value of the function to answer1.

char answer1 = askYesNo1();

This means that answer1 is assigned either ’y’ or ’n’—whichever character the user entered when prompted by askYesNo1().

Next in main(), I display the value of answer1 for all to see.

#### [Accepting Values into Parameters](#_bookmark6)

You can send a function values that it accepts into its parameters. This is the most common way to get information into a function.

***Specifying Parameters***

The second function I declare, askYesNo2(), accepts a value into a parameter. Specifically, it accepts a value of type string. You can tell this from the function prototype before main():

char askYesNo2(string question);

Using Parameters and Return Values 159

**Hint**

You don’t have to use parameter names in a prototype; all you have to include are the parameter types. For example, the following is a perfectly valid prototype that declares askYesNo2(), a function with one string parameter that returns a char.

char askYesNo2(string);

Even though you don’t have to use parameter names in prototypes, it’s a good idea to do so. It makes your code clearer, and it’s worth the minor effort.

From the header of askYesNo2(), you can see that the function accepts a string

object as a parameter and names that parameter question.

char askYesNo2(string question)

Unlike prototypes, you must specify parameter names in a function definition. You use a parameter name inside a function to access the parameter value.

**Trap**

The parameter types specified in a function prototype must match the parameter types listed in the function definition. If they don’t, you’ll generate a nasty compile error.

***Passing Values to Parameters***

The askYesNo2() function is an improvement over askYesNo1(). The new function allows you to ask your own personalized question by passing a string prompt to the function. In main(), I call askYesNo2() with:

char answer2 = askYesNo2("Do you wish to save your game?");

This statement calls askYesNo2() and passes the string literal argument "Do you wish to save your game?" to the function.

***Using Parameter Values***

askYesNo2() accepts "Do you wish to save your game?" into its parameter question, which acts like any other variable in the function. In fact, I display question with:

cout << question << " (y/n): ";

160 Chapter 5 n Functions: Mad Lib

**Hi n t**

Actually, there’s a little more going on behind the scenes here. When the string literal "Do you wish to save your game?" is passed to question, a string object equal to the string literal is created and the string object gets assigned to question.

Just like askYesNo1(), askYesNo2() continues to prompt the user until he enters y or n. Then the function returns that value and ends.

Back in main(), the returned char value is assigned to answer2, which I then display.

#### [Understanding Encapsulation](#_bookmark6)

You might not see the need for return values when you are using your own functions. Why not just use the variables response1 and response2 back in the main()? Because you can’t; response1 and response2 don’t exist outside of the functions in which they were defined. In fact, no variable you create in a function, including its parameters, can be directly accessed outside its function. This is a good thing, and it is called *encapsulation.* Encapsulation helps keep independent code truly separate by hiding or encapsulating the details. That’s why you use parameters and return values—to communicate only the informa- tion that needs to be exchanged. Plus, you don’t have to keep track of variables you create within a function in the rest of your program. As your programs get large, this is a great benefit.

Encapsulation might sound a lot like abstraction. That’s because they’re closely related. Encapsulation is a principal of abstraction. Abstraction saves you from worrying about the details, while encapsulation hides the details from you. As an example, consider a television remote control with volume up and down buttons. When you use a TV remote to change the volume, you’re employing abstraction because you don’t need to know what happens inside the TV for it to work. Now suppose the TV remote has 10 volume levels. You can get to them all through the remote, but you can’t directly access them. That is, you can’t get a specific volume number directly. You can only press the up volume and down volume buttons to eventually get to the level you want. The actual volume number is encapsulated and not directly available to you.

Working with Scopes 161

### [Understanding Software Reuse](#_bookmark6)

You can reuse functions in other programs. For example, since asking the user a yes or no question is such a common thing to do in a game, you could create an askYesNo() function and use it in all of your future game programs. So writing good functions not only saves you time and energy in your current game project, but it can save you effort in future ones, too.

**Real World**

It’s always a waste of time to reinvent the wheel, so *software reuse*—employing existing software and other elements in new projects—is a technique that game companies take to heart. The benefits of software reuse include:

n **Increased company productivity.** By reusing code and other elements that already exist, such as a graphics engine, game companies can get their projects done with less effort.

n **Improved software quality.** If a game company already has a tested piece of code, such as a networking module, then the company can reuse the code with the knowledge that it’s bug-free.

n **Improved software performance.** Once a game company has a high-performance piece of code, using it again not only saves the company the trouble of reinventing the wheel, it saves them from reinventing a less efficient one.

You can reuse code you’ve written by copying from one program and pasting it into another, but there is a better way. You can divide up a big game project into multiple files. You’ll learn about this technique in Chapter 10, “Inheritance and Polymorphism: Blackjack.”

### [Working with Scopes](#_bookmark6)

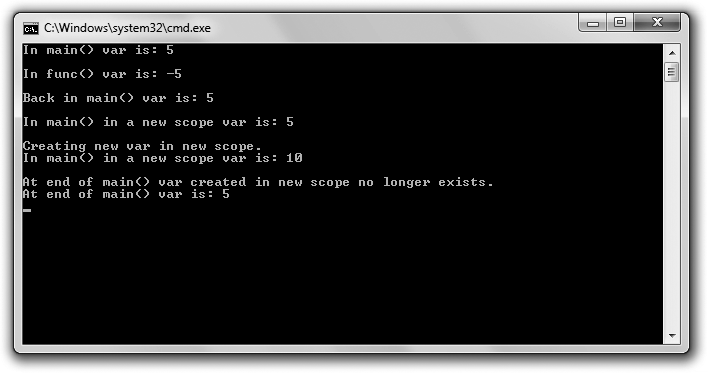
A variable’s *scope* determines where the variable can be seen in your program. Scopes allow you to limit the accessibility of variables and are the key to encapsulation, helping keep separate parts of your program, such as functions, apart from each other.

#### [Introducing the Scoping Program](#_bookmark6)

The Scoping program demonstrates scopes. The program creates three variables with the same name in three separate scopes. The program displays the values of

162 Chapter 5 n Functions: Mad Lib

these variables, and you can see that even though they all have the same name, the variables are completely separate entities. Figure 5.3 shows the results of the program.



**Figure 5.3**

Even though they have the same name, all three variables have a unique existence in their own scopes.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 5 folder; the filename is scoping.cpp.

// Scoping

// Demonstrates scopes #include <iostream> using namespace std; void func();

int main()

{

int var = 5; // local variable in main() cout << "In main() var is: " << var << "\n\n";

Working with Scopes 163

func();

cout << "Back in main() var is: " << var << "\n\n";

{

cout << "In main() in a new scope var is: " << var << "\n\n";

cout << "Creating new var in new scope.\n";

int var = 10; // variable in new scope, hides other variable named var cout << "In main() in a new scope var is: " << var << "\n\n";

}

cout << "At end of main() var created in new scope no longer exists.\n"; cout << "At end of main() var is: " << var << "\n";

return 0;

}

void func()

{

int var = -5; // local variable in func() cout << "In func() var is: " << var << "\n\n";

}

#### [Working with Separate Scopes](#_bookmark6)

Every time you use curly braces to create a block, you create a scope. Functions are one example of this. Variables declared in a scope aren’t visible outside of that scope. This means that variables declared in a function aren’t visible outside of that function.

Variables declared inside a function are considered *local variables*—they’re local to the function. This is what makes functions encapsulated.

You’ve seen many local variables in action already. I define yet another local variable in main() with:

int var = 5; // local variable in main()

This line declares and initializes a local variable named var. I send the variable to cout in the next line of code:

cout << "In main() var is: " << var << "\n\n";

164 Chapter 5 n Functions: Mad Lib

This works just as you’d expect—5 is displayed.

Next I call func(). Once I enter the function, I’m in a separate scope outside of the scope defined by main(). As a result, I can’t access the variable var that I defined in main(). This means that when I next define a variable named var in func() with the following line, this new variable is completely separate from the variable named var in main().

int var = -5; // local variable in func()

The two have no effect on each other, and that’s the beauty of scopes. When you write a function, you don’t have to worry if another function uses the same variable names.

Then, when I display the value of var in func() with the following line, the computer displays -5.

cout << "In func() var is: " << var << "\n\n";

That’s because, as far as the computer can see in this scope, there’s only one variable named var—the local variable I declared in this function.

Once a scope ends, all of the variables declared in that scope cease to exist. They’re said to go *out of scope*. So next, when func() ends, its scope ends. This means all of the variables declared in func() are destroyed. As a result, the var I declared in func() with a value of -5 is destroyed.

After func() ends, control returns to main() and picks up right where it left off. Next, the following line is executed, which sends var to cout.

cout << "Back in main() var is: " << var << "\n\n";

The value of the var local to main() (5) is displayed again.

You might be wondering what happened to the var I created in main() while I was in func(). Well, the variable wasn’t destroyed because main() hadn’t yet ended. (Program control simply took a small detour to func().) When a program momentarily exits one function to enter another, the computer saves its place in the first function, keeping safe the values of all of its local variables, which are reinstated when control returns to the first function.

**Hi n t**

Parameters act just like local variables in functions.

Working with Scopes 165

#### [Working with Nested Scopes](#_bookmark6)

You can create a nested scope with a pair of curly braces in an existing scope. That’s what I do next in main(), with:

{

cout << "In main() in a new scope var is: " << var << "\n\n";

cout << "Creating new var in new scope.\n";

int var = 10; // variable in new scope, hides other variable named var cout << "In main() in a new scope var is: " << var << "\n\n";

}

This new scope is a nested scope in main(). The first thing I do in this nested scope is display var. If a variable hasn’t been declared in a scope, the computer looks up the levels of nested scopes one at a time to find the variable you requested. In this case, because var hasn’t been declared in this nested scope, the computer looks one level up to the scope that defines main() and finds var. As a result, the program displays that variable’s value—5.

However, the next thing I do in this nested scope is declare a new variable named var and initialize it to 10. Now when I send var to cout, 10 is displayed. This time the computer doesn’t have to look up any levels of nested scopes to find var; there’s a var local to this scope. And don’t worry, the var I first declared in main() still exists; it’s simply hidden in this nested scope by the new var.

**Trap**

Although you can declare variables with the same name in a series of nested scopes, it’s not a good idea because it can lead to confusion.

Next, when the nested scope ends, the var that was equal to 10 goes out of scope and ceases to exist. However, the first var I created is still around, so when I display var for the last time in main() with the following line, the program displays 5.

cout << "At end of main() var is: " << var << "\n";

166 Chapter 5 n Functions: Mad Lib

**Hi n t**

When you define variables inside for loops, while loops, if statements, and switch statements, these variables don’t exist outside their structures. They act like variables declared in a nested scope. For example, in the following code, the variable i doesn’t exist outside the loop.

for(int i = 0; i < 10; ++i)

{

cout << i;

}

// i doesn’t exist outside the loop

But beware—some older compilers don’t properly implement this functionality of standard C++. I recommend that you use an IDE with a modern compiler, such as Microsoft Visual C++ 2010 Express Edition. For step-by-step instructions on how to create your first project with this IDE, check out Appendix A.

### [Using Global Variables](#_bookmark6)

Through the magic of encapsulation, the functions you’ve seen are all totally sealed off and independent from each other. The only way to get information into them is through their parameters, and the only way to get information out of them is from their return values. Well, that’s not completely true. There is another way to share information among parts of your program—through *global variables* (variables that are accessible from any part of your program).

#### [Introducing the Global Reach Program](#_bookmark6)

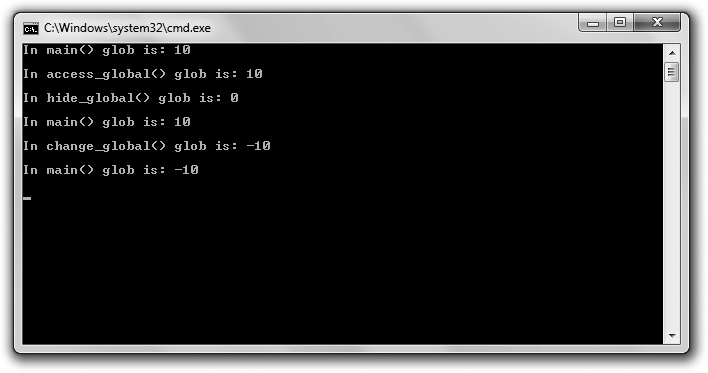
The Global Reach program demonstrates global variables. The program shows how you can access a global variable from anywhere in your program. It also shows how you can hide a global variable in a scope. Finally, it shows that you can change a global variable from anywhere in your program. Figure 5.4 shows the results of the program.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 5 folder; the filename is global\_reach.cpp.

// Global Reach

// Demonstrates global variables #include <iostream>

Using Global Variables 167



**Figure 5.4**

You can access and change global variables from anywhere in a program—but they can also be hidden in a scope as well.

using namespace std;

int glob = 10; // global variable void access\_global();

void hide\_global(); void change\_global();

int main()

{

cout << "In main() glob is: " << glob << "\n\n"; access\_global();

hide\_global();

cout << "In main() glob is: " << glob << "\n\n";

change\_global();

cout << "In main() glob is: " << glob << "\n\n";

return 0;

}

168 Chapter 5 n Functions: Mad Lib

void access\_global()

{

cout << "In access\_global() glob is: " << glob << "\n\n";

}

void hide\_global()

{

int glob = 0; // hide global variable glob

cout << "In hide\_global() glob is: " << glob << "\n\n";

}

void change\_global()

{

glob = -10; // change global variable glob

cout << "In change\_global() glob is: " << glob << "\n\n";

}

#### [Declaring Global Variables](#_bookmark6)

You declare global variables outside of any function in your program file. That’s what I do in the following line, which creates a global variable named glob initialized to 10.

int glob = 10; // global variable

#### [Accessing Global Variables](#_bookmark6)

You can access a global variable from anywhere in your program. To prove it, I display glob in main() with:

cout << "In main() glob is: " << glob << "\n\n";

The program displays 10 because as a global variable, glob is available to any part of the program. To show this again, I next call access\_global(), and the computer executes the following code in that function:

cout << "In access\_global() glob is: " << glob << "\n\n";

Again, 10 is displayed. That makes sense because I’m displaying the exact same variable in each function.

Using Global Variables 169

#### [Hiding Global Variables](#_bookmark7)

You can hide a global variable like any other variable in a scope; you simply declare a new variable with the same name. That’s exactly what I do next, when I call hide\_global(). The key line in that function doesn’t change the global variable glob; instead, it creates a new variable named glob, local to hide\_global(), that hides the global variable.

int glob = 0; // hide global variable glob

As a result, when I send glob to cout next in hide\_global() with the following line, 0 is displayed.

cout << "In hide\_global() glob is: " << glob << "\n\n";

The global variable glob remains hidden in the scope of hide\_global() until the function ends.

To prove that the global variable was only hidden and not changed, next I display glob back in main() with:

cout << "In main() glob is: " << glob << "\n\n";

Once again, 10 is displayed.

**Trap**

Although you can declare variables in a function with the same name as a global variable, it’s not a good idea because it can lead to confusion.

#### [Altering Global Variables](#_bookmark7)

Just as you can access a global variable from anywhere in your program, you can alter one from anywhere in your program, too. That’s what I do next, when I call the change\_global() function. The key line of the function assigns -10 to the global variable glob.

glob = -10; // change global variable glob

To show that it worked, I display the variable in change\_global() with:

cout << "In change\_global() glob is: " << glob << "\n\n";

Then, back in main(), I send glob to cout with:

cout << "In main() glob is: " << glob << "\n\n";

Because the global variable glob was changed, -10 is displayed.

170 Chapter 5 n Functions: Mad Lib

#### [Minimizing the Use of Global Variables](#_bookmark7)

Just because you can doesn’t mean you should. This is a good programming motto. Sometimes things are technically possible, but not a good idea. Using global variables is an example of this. In general, global variables make programs confusing because it can be difficult to keep track of their changing values. You should limit your use of global variables as much as possible.

### [Using Global Constants](#_bookmark7)

Unlike global variables, which can make your programs confusing, *global constants*—constants that can be accessed from anywhere in your program— can help make programs clearer. You declare a global constant much like you declare a global variable—by declaring it outside of any function. And because

you’re declaring a constant, you need to use the const keyword. For example, the following line defines a global constant (assuming the declaration is outside of any function) named MAX\_ENEMIES with a value of 10 that can be accessed anywhere in the program.

const int MAX\_ENEMIES = 10;

**Trap**

Just like with global variables, you can hide a global constant by declaring a local constant with the same name. However, you should avoid this because it can lead to confusion.

How exactly can global constants make game programming code clearer? Well, suppose you’re writing an action game in which you want to limit the total number of enemies that can blast the poor player at once. Instead of using a numeric literal everywhere, such as 10, you could define a global constant MAX\_ENEMIES that’s equal to 10. Then whenever you see that global constant name, you know exactly what it stands for.

One caveat: You should only use global constants if you need a constant value in more than one part of your program. If you only need a constant value in a specific scope (such as in a single function), use a local constant instead.

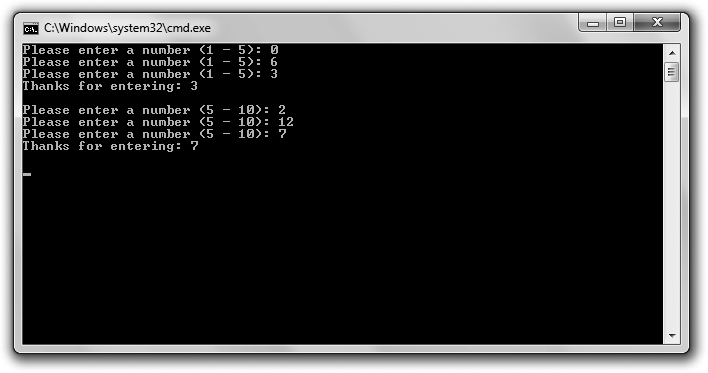
Using Default Arguments 171

### [Using Default Arguments](#_bookmark7)

When you write a function in which a parameter almost always gets passed the same value, you can save the caller the effort of constantly specifying this value by using a *default argument*—a value assigned to a parameter if none is specified. Here’s a concrete example. Suppose you have a function that sets the graphics display. One of your parameters might be bool fullScreen, which tells the function whether to display the game in full-screen or windowed mode. Now, if you think the function will often be called with true for fullScreen, you could give that parameter a default argument of true, saving the caller the effort of passing true to fullScreen whenever the caller invokes this display-setting function.

#### [Introducing the Give Me a Number Program](#_bookmark7)

The Give Me a Number program asks the user for two different numbers in two different ranges. The same function is called each time the user is prompted for a number. However, each call to this function uses a different number of arguments because this function has a default argument for the lower limit. This means the caller can omit an argument for the lower limit, and the function will use a default value automatically. Figure 5.5 shows the results of the program.



**Figure 5.5**

A default argument is used for the lower limit the first time the user is prompted for a number.

172 Chapter 5 n Functions: Mad Lib

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 5 folder; the filename is give\_me\_a\_number.cpp.

// Give Me a Number

// Demonstrates default function arguments

#include <iostream> #include <string>

using namespace std;

int askNumber(int high, int low = 1); int main()

{

int number = askNumber(5);

cout << "Thanks for entering: " << number << "\n\n";

number = askNumber(10, 5);

cout << "Thanks for entering: " << number << "\n\n";

return 0;

}

int askNumber(int high, int low)

{

int num; do

{

cout << "Please enter a number" << " (" << low << " - " << high << "): "; cin >> num;

} while (num > high || num < low);

return num;

}

Using Default Arguments 173

#### [Specifying Default Arguments](#_bookmark7)

The function askNumber() has two parameters—high and low. You can tell this from the function prototype:

int askNumber(int high, int low = 1);

Notice that the second parameter, low, looks like it’s assigned a value. In a way, it is. The 1 is a default argument meaning that if a value isn’t passed to low when the function is called, low is assigned 1. You specify default arguments by using = followed by a value after a parameter name.

**Trap**

Once you specify a default argument in a list of parameters, you must specify default arguments for all remaining parameters. So the following prototype is valid:

void setDisplay(int height, int width, int depth = 32, bool fullScreen = true);

while this one is illegal:

void setDisplay(int width, int height, int depth = 32, bool fullScreen);

By the way, you don’t repeat the default argument in the function definition, as you can see in the function definition of askNumber().

int askNumber(int high, int low)

#### [Assigning Default Arguments to Parameters](#_bookmark7)

The askNumber() function asks the user for a number between an upper and a lower limit. The function keeps asking until the user enters a number within the range, and then it returns the number. I first call the function in main() with:

int number = askNumber(5);

As a result of this code, the parameter high in askNumber() is assigned 5. Because I don’t provide any value for the second parameter, low, it gets assigned the default value of 1. This means the function prompts the user for a number between 1 and 5.

**Trap**

When you are calling a function with default arguments, once you omit an argument, you must omit arguments for all remaining parameters. For example, given the prototype

void setDisplay(int height, int width, int depth = 32, bool fullScreen = true);

174 Chapter 5 n Functions: Mad Lib

a valid call to the function would be

setDisplay(1680, 1050);

while an illegal call would be

setDisplay(1680, 1050, false);

Once the user enters a valid number, askNumber() returns that value and ends. Back in main(), the value is assigned to number and displayed.

#### [Overriding Default Arguments](#_bookmark7)

Next I call askNumber() again with:

number = askNumber(10, 5);

This time I pass a value for low—5. This is perfectly fine; you can pass an argument for any parameter with a default argument, and the value you pass will override the default. In this case, it means that low is assigned 5.

As a result, the user is prompted for a number between 5 and 10. Once the user enters a valid number, askNumber() returns that value and ends. Back in main(), the value is assigned to number and displayed.

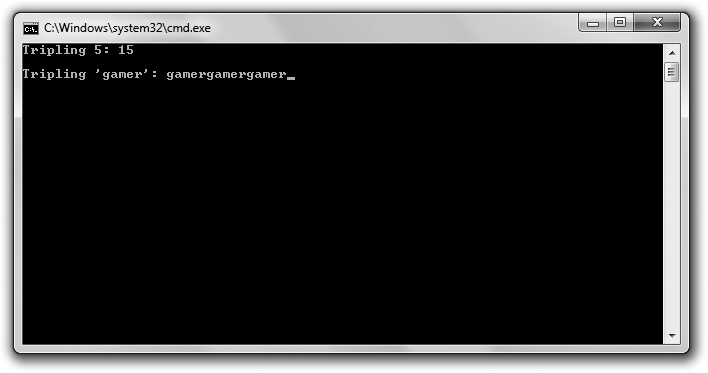
### [Overloading Functions](#_bookmark7)

You’ve seen how you must specify a parameter list and a single return type for each function you write. But what if you want a function that’s more versatile— one that can accept different sets of arguments? For example, suppose you want to write a function that performs a 3D transformation on a set of vertices that are represented as floats, but you want the function to work with ints as well. Instead of writing two separate functions with two different names, you could use function overloading so that a single function could handle the different parameter lists. This way, you could call one function and pass vertices as either floats or ints.

#### [Introducing the Triple Program](#_bookmark7)

The Triple program triples the value 5, and “gamer”. The program triples these values using a single function that’s been overloaded to work with an argument of two different types: int and string. Figure 5.6 shows a sample run of the program.

Overloading Functions 175



**Figure 5.6**

Function overloading allows you to triple the values of two different types using the same function name.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 5 folder; the filename is triple.cpp.

// Triple

// Demonstrates function overloading

#include <iostream> #include <string>

using namespace std; int triple(int number);

string triple(string text);

int main()

{

176 Chapter 5 n Functions: Mad Lib

cout << "Tripling 5: " << triple(5) << "\n\n"; cout << "Tripling ’gamer’: " << triple("gamer");

return 0;

}

int triple(int number)

{

return (number \* 3);

}

string triple(string text)

{

return (text + text + text);

}

#### [Creating Overloaded Functions](#_bookmark7)

To create an overloaded function, you simply need to write multiple function definitions with the same name and different parameter lists. In the Triple program, I write two definitions for the function triple(), each of which specifies a different type as its single argument. Here are the function prototypes:

int triple(int number); string triple(string text);

The first takes an int argument and returns an int. The second takes a string

object and returns a string object.

In each function definition, you can see that I return triple the value sent. In the first function, I return the int sent, tripled. In the second function, I return the string sent, repeated three times.

**Trap**

To implement function overloading, you need to write multiple definitions for the same function with different parameter lists. Notice that I didn’t mention anything about return types. That’s because if you write two function definitions in which only the return type is different, you’ll generate a compile error. For example, you cannot have both of the following prototypes in a program:

int Bonus(int); float Bonus(int);

Inlining Functions 177

#### [Calling Overloaded Functions](#_bookmark7)

You can call an overloaded function the same way you call any other function, by using its name with a set of valid arguments. But with overloaded functions, the compiler (based on the argument values) determines which definition to invoke. For example, when I call triple() with the following line and use an int as the argument, the compiler knows to invoke the definition that takes an int. As a result, the function returns the int 15.

cout << "Tripling 5: " << triple(5) << "\n\n";

I call triple() again with:

cout << "Tripling ’gamer’: " << triple("gamer");

Because I use a string literal as the argument, the compiler knows to invoke the definition of the function that takes a string object. As a result, the function returns the string object equal to gamergamergamer.

### [Inlining Functions](#_bookmark7)

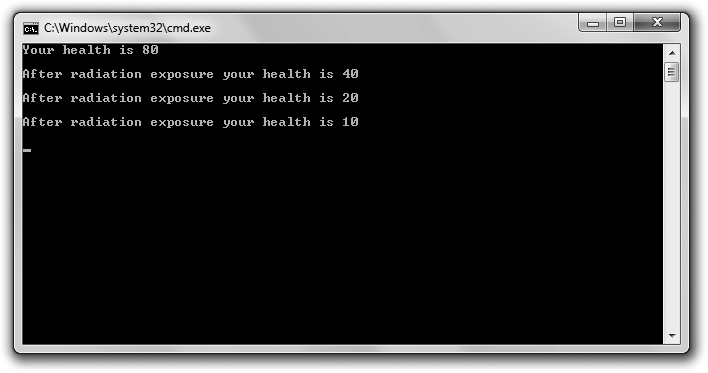
There’s a small performance cost associated with calling a function. Normally, this isn’t a big deal because the cost is relatively minor. However, for tiny functions (such as one or two lines), it’s sometimes possible to speed up program performance by inlining them. By *inlining* a function, you ask the compiler to make a copy of the function everywhere it’s called. As a result, program control doesn’t have to jump to a different location each time the function is called.

#### [Introducing the Taking Damage Program](#_bookmark7)

The Taking Damage program simulates what happens to a character’s health as the character takes radiation damage. The character loses half of his health each round. Fortunately, the program only runs three rounds, so we’re spared the sad end of the character. The program inlines the tiny function that calculates the character’s new health. Figure 5.7 shows the program results.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 5 folder; the filename is taking\_damage.cpp.

178 Chapter 5 n Functions: Mad Lib



**Figure 5.7**

The character approaches his demise quite efficiently as his health decreases through an inlined function.

// Taking Damage

// Demonstrates function inlining #include <iostream>

int radiation(int health); using namespace std;

int main()

{

int health = 80;

cout << "Your health is " << health << "\n\n";

health = radiation(health);

cout << "After radiation exposure your health is " << health << "\n\n";

health = radiation(health);

cout << "After radiation exposure your health is " << health << "\n\n";

health = radiation(health);

cout << "After radiation exposure your health is " << health << "\n\n";

Inlining Functions 179

return 0;

}

inline int radiation(int health)

{

return (health / 2);

}

#### [Specifying Functions for Inlining](#_bookmark7)

To mark a function for inlining, simply put inline before the function  definition. That’s what I do when I define the following function:

inline int radiation(int health)

Note that you don’t use inline in the function declaration:

int radiation(int health);

By flagging the function with inline, you ask the compiler to copy the function directly into the calling code. This saves the overhead of making the function call. That is, program control doesn’t have to jump to another part of your code. For small functions, this can result in a performance boost.

However, inlining is not a silver bullet for performance. In fact, indiscriminate inlining can lead to worse performance because inlining a function creates extra copies of it, which can dramatically increase memory consumption.

**Hint**

When you inline a function, you really make a request to the compiler, which has the ultimate decision on whether to inline the function. If your compiler thinks that inlining won’t boost performance, it won’t inline the function.

#### [Calling Inlined Functions](#_bookmark7)

Calling an inlined function is no different than calling a non-inlined function, as you see with my first call to radiation().

health = radiation(health);

This line of code assigns health one half of its original value.

Assuming that the compiler grants my request for inlining, this code doesn’t result in a function call. Instead, the compiler places the code to halve health

180 Chapter 5 n Functions: Mad Lib

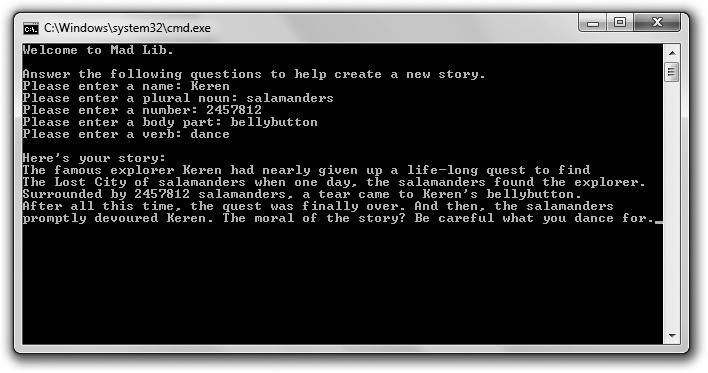
right at this place in the program. In fact, the compiler does this for all three calls to the function.

**Real World**

Although obsessing about performance is a game programmer’s favorite hobby, there’s a danger in focusing too much on speed. In fact, the approach many developers take is to first get their game programs working well before they tweak for small performance gains. At that point, programmers will *profile* their code by running a utility (a profiler) that analyzes where the game program spends its time. If a programmer sees bottlenecks, he or she might consider hand optimizations such as function inlining.

### [Introducing the Mad Lib Game](#_bookmark7)

The Mad Lib game asks for the user’s help in creating a story. The user supplies the name of a person, a plural noun, a number, and a verb. The program takes all of this information and uses it to create a personalized story. Figure 5.8 shows a sample run of the program.



**Figure 5.8**

After the user provides all of the necessary information, the program displays the literary masterpiece.

Introducing the Mad Lib Game 181

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 5 folder; the filename is mad\_lib.cpp.

#### [Setting Up the Program](#_bookmark7)

As usual, I start the program with some comments and include the necessary files.

// Mad-Lib

// Creates a story based on user input

#include <iostream> #include <string>

using namespace std;

string askText(string prompt); int askNumber(string prompt);

void tellStory(string name, string noun, int number, string bodyPart, string verb);

You can tell from my function prototypes that I have three functions in addition to main()—askText(), askNumber(), and tellStory().

#### [The main() Function](#_bookmark7)

The main() function calls all of the other functions. It calls the function askText() to get a name, plural noun, body part, and verb from the user. It calls askNumber() to get a number from the user. It calls tellStory() with all of the user-supplied information to generate and display the story.

int main()

{

cout << "Welcome to Mad Lib.\n\n";

cout << "Answer the following questions to help create a new story.\n";

string name = askText("Please enter a name: ");

string noun = askText("Please enter a plural noun: "); int number = askNumber("Please enter a number: "); string bodyPart = askText("Please enter a body part: "); string verb = askText("Please enter a verb: ");

182 Chapter 5 n Functions: Mad Lib

tellStory(name, noun, number, bodyPart, verb);

return 0;

}

#### [The askText() Function](#_bookmark7)

The askText() function gets a string from the user. The function is versatile and takes a parameter of type string, which it uses to prompt the user. Because of this, I’m able to call this single function to ask the user for a variety of different pieces of information, including a name, plural noun, body part, and verb.

string askText(string prompt)

{

string text; cout << prompt; cin >> text; return text;

}

**Trap**

Remember that this simple use of cin only works with strings that have no white space in them (such as tabs or spaces). So when a user is prompted for a body part, he can enter bellybutton, but medulla oblongata will cause a problem for the program.

There are ways to compensate for this, but that really requires a discussion of something called *streams*, which is beyond the scope of this book. So use cin in this way, but just be aware of its limitations.

#### [The askNumber() Function](#_bookmark7)

The askNumber() function gets an integer from the user. Although I only call it once in the program, it’s versatile because it takes a parameter of type string that it uses to prompt the user.

int askNumber(string prompt)

{

int num;

cout << prompt; cin >> num; return num;

}

Summary 183

#### [The tellStory() Function](#_bookmark7)

The tellStory() function takes all of the information entered by the user and uses it to display a personalized story.

void tellStory(string name, string noun, int number, string bodyPart, string verb)

{

cout << "\nHere’s your story:\n"; cout << "The famous explorer "; cout << name;

cout << " had nearly given up a life-long quest to find\n"; cout << "The Lost City of ";

cout << noun;

cout << " when one day, the "; cout << noun;

cout << " found the explorer.\n"; cout << "Surrounded by ";

cout << number; cout << " " << noun;

cout << ", a tear came to "; cout << name << "’s ";

cout << bodyPart << ".\n";

cout << "After all this time, the quest was finally over. "; cout << "And then, the ";

cout << noun << "\n";

cout << "promptly devoured "; cout << name << ". ";

cout << "The moral of the story? Be careful what you "; cout << verb;

cout << " for.";

}

### [Summary](#_bookmark7)

In this chapter, you should have learned the following concepts:

n Functions allow you to break up your programs into manageable chunks.

n One way to declare a function is to write a function prototype—code that lists the return value, name, and parameter types of a function.

n Defining a function means writing all the code that makes the function tick.

184 Chapter 5 n Functions: Mad Lib

n You can use the return statement to return a value from a function. You can also use return to end a function that has void as its return type.

n A variable’s scope determines where the variable can be seen in your program.

n Global variables are accessible from any part of your program. In general, you should try to limit your use of global variables.

n Global constants are accessible from any part of your program. Using global constants can make your program code clearer.

n Default arguments are assigned to a parameter if no value for the param- eter is specified in the function call.

n Function overloading is the process of creating multiple definitions for the same function, each of which has a different set of parameters.

n Function inlining is the process of asking the compiler to inline a function—meaning that the compiler should make a copy of the function everywhere in the code where the function is called. Inlining very small functions can sometimes yield a performance boost.

### [Questions and Answers](#_bookmark7)

Q: Why should I write functions?

A: Functions allow you to break up your programs into logical pieces. These pieces result in smaller, more manageable chunks of code, which are easier to work with than a single monolithic program.

Q: What’s encapsulation?

A: At its core, encapsulation is about keeping things separate. Function encapsulation provides that variables declared in a function are not accessible outside the function, for example.

Q: What’s the difference between an argument and a parameter?

A: An argument is what you use in a function call to pass a value to a function. A parameter is what you use in a function definition to accept values passed to a function.

Questions and Answers 185

Q: Can I have more than one return statement in a function?

A: Sure. In fact, you might want multiple return statements to specify different end points of a function.

Q: What’s a local variable?

A: A variable that’s defined in a scope. All variables defined in a function are local variables; they’re local to that function.

Q: What does it mean to hide a variable?

A: A variable is hidden when you declare it inside a new scope with the same name as a variable in an outer scope. As a result, you can’t get to the variable in the outer scope by using its variable name in the inner scope.

Q: When does a variable go out of scope?

A: A variable goes out of scope when the scope in which it was created ends. Q: What does it mean when a variable goes out of scope?

A: It means the variable ceases to exist. Q: What’s a nested scope?

A: A scope created within an existing scope.

Q: Must an argument have the same name as the parameter to which it’s passed?

A: No. You’re free to use different names. It’s only the value that’s passed from a function call to a function.

Q: Can I write one function that calls another?

A: Of course. In fact, whenever you write a function that you call from main(), you’re doing just that. In addition, you can write a function (other than main()) that calls another function.

Q: What is code profiling?

A: It’s the process of recording how much CPU time various parts of a program use.

186 Chapter 5 n Functions: Mad Lib

Q: Why profile code?

A: To determine any bottlenecks in a program. Sometimes it makes sense to revisit these sections of code in an attempt to optimize them.

Q: When do programmers profile code?

A: Usually toward the end of the programming of a game project. Q: What’s premature optimization?

A: An attempt to optimize code too early in the development process. Code optimization usually makes sense near the end of programming a game project.

### [Discussion Questions](#_bookmark7)

1. How does function encapsulation help you write better programs?
2. How can global variables make code confusing?
3. How can global constants make code clearer?
4. What are the pros and cons of optimizing code?
5. How can software reuse benefit the game industry?

### [Exercises](#_bookmark7)

1. What’s wrong with the following prototype?

int askNumber(int low = 1, int high);

1. Rewrite the Hangman game from Chapter 4 using functions. Include a function to get the player’s guess and another function to determine whether the player’s guess is in the secret word.
2. Using default arguments, write a function that asks the user for a number and returns that number. The function should accept a string prompt from the calling code. If the caller doesn’t supply a string for the prompt, the function should use a generic prompt. Next, using function overloading, write a function that achieves the same results.

|  |  |
| --- | --- |
|  |  |
| [chapter 6](#_bookmark7) | |

# [References: Tic-Tac-Toe](#_bookmark7)

The concept of references is simple, but its implications are profound. In this chapter, you’ll learn about references and how they can help you write more efficient game code. Specifically, you’ll learn to:

n Create references

n Access and change referenced values

n Pass references to functions to alter argument values or for efficiency

n Return references from a function for efficiency or to alter values

### [Using References](#_bookmark7)

A *reference* provides another name for a variable. Whatever you do to a reference is done to the variable to which it refers. You can think of a reference as a nickname for a variable—another name that the variable goes by. In the first program in this chapter, I’ll show you how to create references. Then, in the next few programs, I’ll show you why you’d want to use references and how they can improve your game programs.

#### [Introducing the Referencing Program](#_bookmark7)

The Referencing program demonstrates references. The program declares and initializes a variable to hold a score and then creates a reference that refers to the variable. The program displays the score using the variable and the reference to

187

188 Chapter 6 n References: Tic-Tac-Toe

show that they access the same single value. Next, the program shows that this single value can be altered through either the variable or the reference. Figure 6.1 illustrates the program.



**Figure 6.1**

The variable myScore and the reference mikesScore are both names for the single score value.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 6 folder; the filename is referencing.cpp.

// Referencing

// Demonstrates using references #include <iostream>

using namespace std;

int main()

{

int myScore = 1000;

int& mikesScore = myScore; //create a reference

cout << "myScore is: " << myScore << "\n";

cout << "mikesScore is: " << mikesScore << "\n\n";

Using References 189

cout << "Adding 500 to myScore\n"; myScore += 500;

cout << "myScore is: " << myScore << "\n";

cout << "mikesScore is: " << mikesScore << "\n\n";

cout << "Adding 500 to mikesScore\n"; mikesScore += 500;

cout << "myScore is: " << myScore << "\n";

cout << "mikesScore is: " << mikesScore << "\n\n";

return 0;

}

#### [Creating References](#_bookmark7)

The first thing I do in main() is create a variable to hold my score.

int myScore = 1000;

Then I create a reference that refers to myScore.

int& mikesScore = myScore; //create a reference

The preceding line declares and initializes mikesScore, a reference that refers to myScore. mikesScore is an alias for myScore. mikesScore does not hold its own int value; it’s simply another way to get at the int value that myScore holds.

To declare and initialize a reference, start with the type of value to which the reference will refer, followed by the reference operator (&), followed by the reference name, followed by =, followed by the variable to which the reference will refer.

**Trick**

Sometimes programmers prefix a reference name with the letter “r” to remind them that they’re working with a reference. A programmer might include the following lines:

int playerScore = 1000; int& rScore = playerScore;

One way to understand references is to think of them as nicknames. For example, suppose you’ve got a friend named Eugene, and he (understandably) asks to be called by a nickname—Gibby (not much of an improvement, but it’s what Eugene

190 Chapter 6 n References: Tic-Tac-Toe

wants). So when you’re at a party with your friend, you can call him over using either Eugene or Gibby. Your friend is only one person, but you can call him using either his name or a nickname. This is the same way a variable and a reference to that variable work. You can get to a single value stored in a variable by using its variable name or the name of a reference to that variable. Finally, whatever you do, try not to name your variables Eugene—for their sakes.

**Trap**

Because a reference must always refer to another value, you must initialize the reference when you declare it. If you don’t, you’ll get a compile error. The following line is quite illegal:

int& mikesScore; //don’t try this at home!

#### [Accessing Referenced Values](#_bookmark7)

Next I send both myScore and mikesScore to cout.

cout << "myScore is: " << myScore << "\n";

cout << "mikesScore is: " << mikesScore << "\n\n";

Both lines of code display 1000 because they each access the same single chunk of memory that stores the number 1000. Remember, there is only one value, and it is stored in the variable myScore. mikesScore simply provides another way to get to that value.

#### [Altering Referenced Values](#_bookmark7)

Next I increase the value of myScore by 500.

myScore += 500;

When I send myScore to cout, 1500 is displayed, just as you’d expect. When I send mikesScore to cout, 1500 is also displayed. Again, that’s because mikesScore is just another name for the variable myScore. In essence, I’m sending the same variable to cout both times.

Next I increase mikesScore by 500.

mikesScore += 500;

Because mikesScore is just another name for myScore, the preceding line of code increases the value of myScore by 500. So when I next send myScore to cout, 2000 is displayed. When I send mikesScore to cout, 2000 is displayed again.

Passing References to Alter Arguments 191

**Trap**

A reference always refers to the variable with which it was initialized. You can’t reassign a reference to refer to another variable so, for example, the results of the following code might not be obvious.

int myScore = 1000;

int& mikesScore = myScore; int larrysScore = 2500;

mikesScore = larrysScore; //may not do what you think!

The line mikesScore = larrysScore; does not reassign the reference mikesScore so it refers to larrysScore because a reference can’t be reassigned. However, because mikesScore is just another name for myScore, the code mikesScore = larrysScore; is equivalent to myScore = larrysScore;, which assigns 2500 to myScore. And after all is said and done, myScore becomes 2500 and mikesScore still refers to myScore.

### [Passing References to Alter Arguments](#_bookmark7)

Now that you’ve seen how references work, you might be wondering why you’d ever use them. Well, references come in quite handy when you are passing variables to functions because when you pass a variable to a function, the function gets a copy of the variable. This means that the original variable you passed (called the *argument variable*) can’t be changed. Sometimes this might be exactly what you want because it keeps the argument variable safe and unalterable. But other times you might want to change an argument variable from inside the function to which it was passed. You can accomplish this by using references.

#### [Introducing the Swap Program](#_bookmark7)

The Swap program defines two variables—one that holds my pitifully low score and another that holds your impressively high score. After displaying the scores, the program calls a function meant to swap the scores. But because only copies of the score values are sent to the function, the argument variables that hold the scores are unchanged. Next, the program calls another swap function. This time, through the use of references, the argument variables’ values are successfully exchanged—giving me the great big score and leaving you with the small one. Figure 6.2 shows the program in action.

192 Chapter 6 n References: Tic-Tac-Toe



**Figure 6.2**

Passing references allows goodSwap() to alter the argument variables.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 6 folder; the filename is swap.cpp.

// Swap

// Demonstrates passing references to alter argument variables #include <iostream>

using namespace std;

void badSwap(int x, int y); void goodSwap(int& x, int& y);

int main()

{

int myScore = 150; int yourScore = 1000;

cout << "Original values\n";

cout << "myScore: " << myScore << "\n";

cout << "yourScore: " << yourScore << "\n\n";

Passing References to Alter Arguments 193

cout << "Calling badSwap()\n"; badSwap(myScore, yourScore);

cout << "myScore: " << myScore << "\n";

cout << "yourScore: " << yourScore << "\n\n";

cout << "Calling goodSwap()\n"; goodSwap(myScore, yourScore);

cout << "myScore: " << myScore << "\n"; cout << "yourScore: " << yourScore << "\n";

return 0;

}

void badSwap(int x, int y)

{

int temp = x; x = y;

y = temp;

}

void goodSwap(int& x, int& y)

{

int temp = x; x = y;

y = temp;

}

#### [Passing by Value](#_bookmark7)

After declaring and initializing myScore and yourScore, I send them to cout. As you’d expect, 150 and 1000 are displayed. Next I call badSwap().

When you specify a parameter the way you’ve seen so far (as an ordinary variable, not as a reference), you’re indicating that the argument for that parameter will be *passed by value*, meaning that the parameter will get a *copy* of the argument variable and not access to the argument variable itself. By looking at the function header of badSwap(), you can tell that a call to the function passes both arguments by value.

void badSwap(int x, int y)

194 Chapter 6 n References: Tic-Tac-Toe

This means that when I call badSwap() with the following line, copies of myScore

and yourScore are sent to the parameters, x and y.

badSwap(myScore, yourScore);

Specifically, x is assigned 150 and y is assigned 1000. As a result, nothing I do with x and y in the function badSwap() will have any effect on myScore and yourScore.

When the guts of badSwap() execute, x and y do exchange values—x becomes 1000 and y becomes 150. However, when the function ends, both x and y go out of scope and cease to exist. Control then returns to main(), where myScore and yourScore haven’t changed. Then, when I send myScore and yourScore to cout, 150 and 1000 are displayed again. Sadly, I still have the small score and you still have the large one.

#### [Passing by Reference](#_bookmark7)

It’s possible to give a function access to an argument variable by passing a parameter a reference to the argument variable. As a result, anything done to the parameter will be done to the argument variable. To *pass by reference*, you must first declare the parameter as a reference.

You can tell that a call to goodSwap() passes both arguments by reference by looking at the function header.

void goodSwap(int& x, int& y)

This means that when I call goodSwap() with the following line, the parameter x

will refer to myScore and the parameter y will refer to yourScore.

goodSwap(myScore, yourScore);

This means that x is just another name for myScore and y is just another name for yourScore. When goodSwap() executes and x and y exchange values, what really happens is that myScore and yourScore exchange values.

After the function ends, control returns to main(), where I send myScore and yourScore to cout. This time 1000 and 150 are displayed. The variables have exchanged values. I’ve taken the large score and left you with the small one. Success at last!

Passing References for Efficiency 195

### [Passing References for Efficiency](#_bookmark7)

Passing a variable by value creates some overhead because you must copy the variable before you assign it to a parameter. When we’re talking about variables of simple, built-in types, such as an int or a float, the overhead is negligible. But a large object, such as one that represents an entire 3D world, could be expensive to copy. Passing by reference, on the other hand, is efficient because you don’t make a copy of an argument variable. Instead, you simply provide access to the existing object through a reference.

#### [Introducing the Inventory Displayer Program](#_bookmark7)

The Inventory Displayer program creates a vector of strings that represents a hero’s inventory. The program then calls a function that displays the inventory. The program passes the displayer function the vector of items as a reference, so it’s an efficient call; the vector isn’t copied. However, there’s a new wrinkle. The program passes the vector as a special kind of reference that prohibits the displayer function from changing the vector. Figure 6.3 shows you the program.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 6 folder; the filename is inventory\_displayer.cpp.



**Figure 6.3**

The vector inventory is passed in a safe and efficient way to the function that displays the hero’s items.

196 Chapter 6 n References: Tic-Tac-Toe

// Inventory Displayer

// Demonstrates constant references

#include <iostream> #include <string> #include <vector>

using namespace std;

//parameter vec is a constant reference to a vector of strings void display(const vector<string>& inventory);

int main()

{

vector<string> inventory; inventory.push\_back("sword"); inventory.push\_back("armor"); inventory.push\_back("shield");

display(inventory);

return 0;

}

//parameter vec is a constant reference to a vector of strings void display(const vector<string>& vec)

{

cout << "Your items:\n";

for (vector<string>::const\_iterator iter = vec.begin(); iter != vec.end(); ++iter)

{

cout << \*iter << endl;

}

}

#### [Understanding the Pitfalls of Reference Passing](#_bookmark8)

One way to efficiently give a function access to a large object is to pass it by reference. However, this introduces a potential problem. As you saw in the Swap program, it opens up an argument variable to being changed. But what if you

Passing References for Efficiency 197

don’t want to change the argument variable? Is there a way to take advantage of the efficiency of passing by reference while protecting an argument variable’s integrity? Yes, there is. The answer is to pass a constant reference.

**Hint**

In general, you should avoid changing an argument variable. Try to write functions that send back new information to the calling code through a return value.

#### [Declaring Parameters as Constant References](#_bookmark8)

The function display() shows the contents of the hero’s inventory. In the function’s header I specify one parameter—a constant reference to a vector of string objects named vec.

void display(const vector<string>& vec)

A *constant reference* is a restricted reference. It acts like any other reference, except you can’t use it to change the value to which it refers. To create a constant reference, simply put the keyword const before the type in the reference declaration.

What does this all mean for the function display()? Because the parameter vec is a constant reference, it means display() can’t change vec. In turn, this means that inventory is safe; it can’t be changed by display(). In general, you can efficiently pass an argument to a function as a constant reference so it’s accessible, but not changeable. It’s like providing the function read-only access to the argument. Although constant references are very useful for specifying function parameters, you can use them anywhere in your program.

**Hint**

A constant reference comes in handy in another way. If you need to assign a constant value to a reference, you have to assign it to a constant reference. (A non-constant reference won’t do.)

#### [Passing a Constant Reference](#_bookmark8)

Back in main(), I create inventory and then call display() with the following line, which passes the vector as a constant reference.

display(inventory);

198 Chapter 6 n References: Tic-Tac-Toe

This results in an efficient and safe function call. It’s efficient because only a reference is passed; the vector is not copied. It’s safe because the reference to the vector is a constant reference; inventory can’t be changed by display().

**Trap**

You can’t modify a parameter marked as a constant reference. If you try, you’ll generate a compile error.

Next, display() lists the elements in the vector using a constant reference to

inventory. Then control returns to main() and the program ends.

### [Deciding How to Pass Arguments](#_bookmark8)

At this point you’ve seen three different ways to pass arguments—by value, as a reference, and as a constant reference. So how do you decide which method to use? Here are some guidelines:

n **By value.** Pass by value when an argument variable is one of the funda- mental built-in types, such as bool, int, or float. Objects of these types are so small that passing by reference doesn’t result in any gain in effi- ciency. You should also pass by value when you want the computer to make a copy of a variable. You might want to use a copy if you plan to alter a parameter in a function, but you don’t want the actual argument variable to be affected.

n **As a constant reference.** Pass a constant reference when you want to efficiently pass a value that you don’t need to change.

n **As a reference.** Pass a reference only when you want to alter the value of the argument variable. However, you should try to avoid changing argument variables whenever possible.

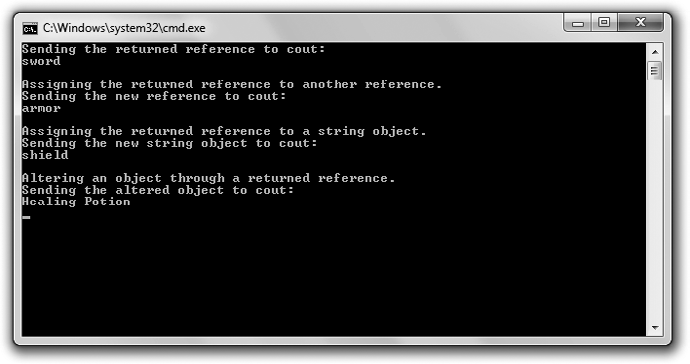
### [Returning References](#_bookmark8)

Just like when you pass a value, when you return a value from a function, you’re really returning a copy of the value. Again, for values of the basic built-in types, this isn’t a big deal. However, it can be an expensive operation if you’re returning a large object. Returning a reference is an efficient alternative.

Returning References 199

#### [Introducing the Inventory Referencer Program](#_bookmark8)

The Inventory Referencer program demonstrates returning references. The program displays the elements of a vector that holds a hero’s inventory by using returned references. Then the program changes one of the items through a returned reference. Figure 6.4 shows the results of the program.



**Figure 6.4**

The items in the hero’s inventory are displayed and changed by using returned references.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 6 folder; the filename is inventory\_referencer.cpp.

// Inventory Referencer

// Demonstrates returning a reference

#include <iostream> #include <string> #include <vector>

using namespace std;

//returns a reference to a string

string& refToElement(vector<string>& inventory, int i);

200 Chapter 6 n References: Tic-Tac-Toe

int main()

{

vector<string> inventory; inventory.push\_back("sword"); inventory.push\_back("armor"); inventory.push\_back("shield");

//displays string that the returned reference refers to cout << "Sending the returned reference to cout:\n"; cout << refToElement(inventory, 0) << "\n\n";

//assigns one reference to another -- inexpensive assignment

cout << "Assigning the returned reference to another reference.\n"; string& rStr = refToElement(inventory, 1);

cout << "Sending the new reference to cout:\n"; cout << rStr << "\n\n";

//copies a string object -- expensive assignment

cout << "Assigning the returned reference to a string object.\n"; string str = refToElement(inventory, 2);

cout << "Sending the new string object to cout:\n"; cout << str << "\n\n";

//altering the string object through a returned reference cout << "Altering an object through a returned reference.\n"; rStr = "Healing Potion";

cout << "Sending the altered object to cout:\n"; cout << inventory[1] << endl;

return 0;

}

//returns a reference to a string

string& refToElement(vector<string>& vec, int i)

{

return vec[i];

}

#### [Returning a Reference](#_bookmark8)

Before you can return a reference from a function, you must specify that you’re returning one. That’s what I do in the refToElement() function header.

string& refToElement(vector<string>& inventory, int i)

Returning References 201

By using the reference operator in string& when I specify the return type, I’m saying that the function will return a reference to a string object (not a string object itself). You can use the reference operator like I did to specify that a function returns a reference to an object of a particular type. Simply put the reference operator after the type name.

The body of the function refToElement() contains only one statement, which returns a reference to the element at position i in the vector.

return vec[i];

Notice that there’s nothing in the return statement to indicate that the function returns a reference. The function header and prototype determine whether a function returns an object or a reference to an object.

**Trap**

Although returning a reference can be an efficient way to send information back to a calling function, you have to be careful not to return a reference to an out-of-scope object—an object that ceases to exist. For example, the following function returns a reference to a string object that no longer exists after the function ends—and that’s illegal.

string& badReference()

{

string local = "This string will cease to exist once the function ends."; return local;

}

One way to avoid this type of problem is to never return a reference to a local variable.

#### [Displaying the Value of a Returned Reference](#_bookmark8)

After creating inventory, a vector of items, I display the first item through a returned reference.

cout << refToElement(inventory, 0) << "\n\n";

The preceding code calls refToElement(), which returns a reference to the element at position 0 of inventory and then sends that reference to cout. As a result, sword is displayed.

202 Chapter 6 n References: Tic-Tac-Toe

#### [Assigning a Returned Reference to a Reference](#_bookmark8)

Next I assign a returned reference to another reference with the following line, which takes a reference to the element in position 1 of inventory and assigns it to rStr.

string& rStr = refToElement(inventory, 1);

This is an efficient assignment because assigning a reference to a reference does not involve the copying of an object. Then I send rStr to cout, and armor is displayed.

#### [Assigning a Returned Reference to a Variable](#_bookmark8)

Next I assign a returned reference to a variable.

string str = refToElement(inventory, 2);

The preceding code doesn’t assign a reference to str. It can’t, because str is a string object. Instead, the code copies the element to which the returned reference refers (the element in position 2 of inventory) and assigns that new copy of the string object to str. Because this kind of assignment involves copying an object, it’s more expensive than assigning one reference to another. Sometimes the cost of copying an object this way is perfectly acceptable, but you should be aware of the extra overhead associated with this kind of assignment and avoid it when necessary.

Next I send the new string object, str, to cout, and shield is displayed.

#### [Altering an Object through a Returned Reference](#_bookmark8)

You can also alter the object to which a returned reference refers. This means you can change the hero’s inventory through rStr, as in the following line of code.

rStr = "Healing Potion";

Because rStr refers to the element in position 1 of inventory, this code changes inventory[1] so it’s equal to "Healing Potion". To prove it, I display the element using the following line, which does indeed show Healing Potion.

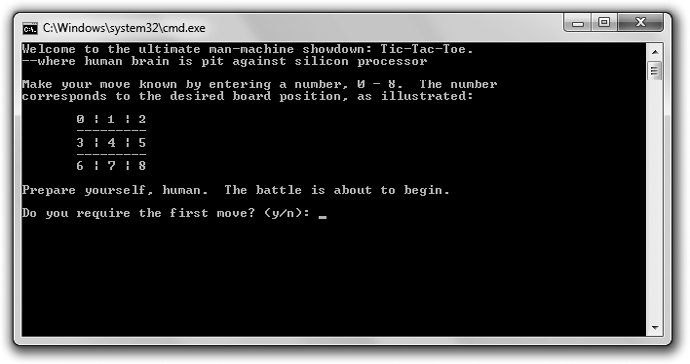
cout << inventory[1] << endl;

If I want to protect inventory so a reference returned by refToElement() can’t be used to change the vector, I should specify the return type of the function as a constant reference.

Introducing the Tic-Tac-Toe Game 203

### [Introducing the Tic-Tac-Toe Game](#_bookmark8)

In this chapter project, you’ll learn how to create a computer opponent using a dash of AI (*Artificial Intelligence*). In the game, the player and computer square off in a high-stakes, man-versus-machine showdown of Tic-Tac-Toe. The computer plays a formidable (although not perfect) game and comes with enough attitude to make any match fun. Figure 6.5 shows the start of a match.



**Figure 6.5**

The computer is full of.. .confidence.

#### [Planning the Game](#_bookmark8)

This game is your most ambitious project yet. You certainly have all the skills you need to create it, but I’m going to go through a longer planning section to help you get the big picture and understand how to create a larger program. Remember, the most important part of programming is planning to program. Without a roadmap, you’ll never get to where you want to go (or it’ll take you a lot longer as you travel the scenic route).

**Real World**

Game designers work countless hours on concept papers, design documents, and prototypes before programmers write any game code. Once the design work is done, the programmers start their work— more planning. It’s only after programmers write their own technical designs that they then begin coding in earnest. The moral of this story? Plan. It’s easier to scrap a blueprint than a 50-story building.

204 Chapter 6 n References: Tic-Tac-Toe

***Writing the Pseudocode***

It’s back to your favorite language that’s not really a language—pseudocode. Because I’ll be using functions for most of the tasks in the program, I can afford to think about the code at a pretty abstract level. Each line of pseudocode should feel like one function call. Later, all I’ll have to do is write the functions that the plan implies. Here’s the pseudocode:

Create an empty Tic-Tac-Toe board Display the game instructions Determine who goes first

Display the board

While nobody’s won and it’s not a tie If it’s the human’s turn

Get the human’s move

Update the board with the human’s move Otherwise

Calculate the computer’s move

Update the board with the computer’s move Display the board

Switch turns

Congratulate the winner or declare a tie

***Representing the Data***

All right, I’ve got a good plan, but it is pretty abstract and talks about throwing around different elements that aren’t really defined in my mind yet. I see the idea of making a move as placing a piece on a game board. But how exactly am I going to represent the game board? Or a piece? Or a move?

Since I’m going to display the game board on the screen, why not just represent a piece as a single character—an X or an O? An empty piece could just be a space. Therefore, the board itself could be a vector of chars. There are nine squares on a Tic-Tac-Toe board, so the vector should have nine elements. Each square on the board will correspond to an element in the vector. Figure 6.6 illustrates what I mean.

Each square or position on the board is represented by a number, 0–8. That means the vector will have nine elements, giving it position numbers 0–8. Because each move indicates a square where a piece should be placed, a move is also just a number, 0–8. That means a move could be represented as an int.

Introducing the Tic-Tac-Toe Game 205

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

**Figure 6.6**

Each square number corresponds to a position in the vector that represents the board.

The side the player and computer play could also be represented by a char— either an ’X’ or an ’O’, just like a game piece. A variable to represent the side of the current turn would also be a char, either an ’X’ or an ’O’.

***Creating a List of Functions***

The pseudocode inspires the different functions I’ll need. I created a list of them, thinking about what each will do, what parameters they’ll have, and what values they’ll return. Table 6.1 shows the results of my efforts.

#### [Setting Up the Program](#_bookmark8)

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 6 folder; the filename is tic-tac-toe.cpp. I’ll go over the code here, section by section.

The first thing I do in the program is include the files I need, define some global constants, and write my function prototypes.

// Tic-Tac-Toe

// Plays the game of tic-tac-toe against a human opponent

#include <iostream> #include <string> #include <vector> #include <algorithm>

using namespace std;

206 Chapter 6 n References: Tic-Tac-Toe

**Table 6.1** Tic-Tac-Toe Functions

**Function Description**

void instructions() Displays the game instructions.

char askYesNo(string *question*) Asks a yes or no question. Receives a question. Returns

either a ’y’ or an ’n’.

int askNumber(string *question*, int

*high*, int *low* = 0)

Asks for a number within a range. Receives a question, a low number, and a high number. Returns a number in the range from *low* to *high*.

char humanPiece() Determines the human’s piece. Returns either an ’X’ or an ’O’.

char opponent(char *piece*) Calculates the opposing piece given a piece. Receives

either an ’X’ or an ’O’. Returns either an ’X’ or an

’O’.

void displayBoard(const vector<char>& *board*)

char winner(const vector<char>&

*board*)

bool isLegal(const vector<char>&

*board*, int *move*)

int humanMove(const vector<char>&

*board*, char *human*)

int computerMove(vector<char> *board*, char *computer*)

void announceWinner(char *winner*, char *computer*, char *human*)

Displays the board on the screen. Receives a board.

Determines the game winner. Receives a board. Returns an ’X’, ’O’, ’T’ (to indicate a tie), or ’N’ (to indicate that no one has won yet).

Determines whether a move is legal. Receives a board and a move. Returns either true or false.

Gets the human’s move. Receives a board and the human’s piece. Returns the human’s move.

Calculates the computer’s move. Receives a board and the computer’s piece. Returns the computer’s move. Congratulates the winner or declares a tie. Receives the

winning side, the computer’s piece, and the human’s

piece.

// global constants const char X = ’X’; const char O = ’O’; const char EMPTY = ’ ’; const char TIE = ’T’; const char NO\_ONE = ’N’;

// function prototypes void instructions();

char askYesNo(string question);

Introducing the Tic-Tac-Toe Game 207

int askNumber(string question, int high, int low = 0); char humanPiece();

char opponent(char piece);

void displayBoard(const vector<char>& board); char winner(const vector<char>& board);

bool isLegal(const vector<char>& board, int move); int humanMove(const vector<char>& board, char human); int computerMove(vector<char> board, char computer);

void announceWinner(char winner, char computer, char human);

In the global constants section, X is shorthand for the char ’X’, one of the two pieces in the game. O represents the char ’O’, the other piece in the game. EMPTY, also a char, represents an empty square on the board. It’s a space because when it’s displayed, it will look like an empty square. TIE is a char that represents a tie game. And NO\_ONE is a char used to represent neither side of the game, which I use to indicate that no one has won yet.

#### [The main() Function](#_bookmark8)

As you can see, the main() function is almost exactly the pseudocode I created earlier.

// main function int main()

{

int move;

const int NUM\_SQUARES = 9;

vector<char> board(NUM\_SQUARES, EMPTY);

instructions();

char human = humanPiece();

char computer = opponent(human); char turn = X; displayBoard(board);

while (winner(board) == NO\_ONE)

{

if (turn == human)

{

move = humanMove(board, human); board[move] = human;

208 Chapter 6 n References: Tic-Tac-Toe

}

else

{

}

move = computerMove(board, computer); board[move] = computer;

displayBoard(board); turn = opponent(turn);

}

announceWinner(winner(board), computer, human);

return 0;

}

#### [The instructions() Function](#_bookmark8)

This function displays the game instructions and gives the computer opponent a little attitude.

void instructions()

{

cout << "Welcome to the ultimate man-machine showdown: Tic-Tac-Toe.\n"; cout << "--where human brain is pit against silicon processor\n\n";

cout << "Make your move known by entering a number, 0 - 8. The number\n"; cout << "corresponds to the desired board position, as illustrated:\n\n";

cout << " 0 | 1 | 2\n";

cout << " ——————— \n";

cout << " 3 | 4 | 5\n";

cout << " ——————— \n";

cout << " 6 | 7 | 8\n\n";

cout << "Prepare yourself, human. The battle is about to begin.\n\n";

}

#### [The askYesNo() Function](#_bookmark8)

This function asks a yes or no question. It keeps asking the question until the player enters either a y or an n. It receives a question and returns either a ’y’ or an ’n’.

Introducing the Tic-Tac-Toe Game 209

char askYesNo(string question)

{

char response; do

{

cout << question << " (y/n): "; cin >> response;

} while (response != ’y’ && response != ’n’);

return response;

}

#### [The askNumber() Function](#_bookmark8)

This function asks for a number within a range and keeps asking until the player enters a valid number. It receives a question, a high number, and a low number. It returns a number within the range specified.

int askNumber(string question, int high, int low)

{

int number; do

{

cout << question << " (" << low << " - " << high << "): "; cin >> number;

} while (number > high || number < low);

return number;

}

If you take a look at this function’s prototype, you can see that the low number has a default value of 0. I take advantage of this fact when I call the function later in the program.

#### [The humanPiece() Function](#_bookmark8)

This function asks the player if he wants to go first, and returns the human’s piece based on that choice. As the great tradition of Tic-Tac-Toe dictates, the X goes first.

char humanPiece()

{

210 Chapter 6 n References: Tic-Tac-Toe

char go\_first = askYesNo("Do you require the first move?"); if (go\_first == ’y’)

{

cout << "\nThen take the first move. You will need it.\n"; return X;

}

else

{

cout << "\nYour bravery will be your undoing... I will go first.\n"; return O;

}

}

#### [The opponent() Function](#_bookmark8)

This function gets a piece (either an ’X’ or an ’O’) and returns the opponent’s piece (either an ’X’ or an ’O’).

char opponent(char piece)

{

if (piece == X)

{

return O;

}

else

{

return X;

}

}

#### [The displayBoard() Function](#_bookmark8)

This function displays the board passed to it. Because each element in the board is either a space, an ’X’, or an ’O’, the function can display each one. I use a few other characters on my keyboard to draw a decent-looking Tic-Tac-Toe board.

void displayBoard(const vector<char>& board)

{

cout << "\n\t" << board[0] << " | " << board[1] << " | " << board[2]; cout << "\n\t" << "———————";

cout << "\n\t" << board[3] << " | " << board[4] << " | " << board[5]; cout << "\n\t" << "———————";

Introducing the Tic-Tac-Toe Game 211

cout << "\n\t" << board[6] << " | " << board[7] << " | " << board[8]; cout << "\n\n";

}

Notice that the vector that represents the board is passed through a constant reference. This means that the vector is passed efficiently; it is not copied. It also means that the vector is safeguarded against any changes. Since I plan to simply display the board and not change it in this function, this is perfect.

#### [The winner() Function](#_bookmark8)

This function receives a board and returns the winner. There are four possible values for a winner. The function will return either X or O if one of the players has won. If every square is filled and no one has won, it returns TIE. Finally, if no one has won and there is at least one empty square, the function returns NO\_ONE.

char winner(const vector<char>& board)

{

// all possible winning rows

const int WINNING\_ROWS[8][3] = { {0, 1, 2},

{3, 4, 5},

{6, 7, 8},

{0, 3, 6},

{1, 4, 7},

{2, 5, 8},

{0, 4, 8},

{2, 4, 6} };

The first thing to notice is that the vector that represents the board is passed through a constant reference. This means that the vector is passed efficiently; it is not copied. It also means that the vector is safeguarded against any change.

In this initial section of the function, I define a constant, two-dimensional array of ints called WINNING\_ROWS, which represents all eight ways to get three in a row and win the game. Each winning row is represented by a group of three numbers—three board positions that form a winning row. For example, the group {0, 1, 2} represents the top row—board positions 0, 1, and 2. The next

group, {3, 4, 5}, represents the middle row—board positions 3, 4, and 5. And so on... .

Next I check to see whether either player has won.

212 Chapter 6 n References: Tic-Tac-Toe

const int TOTAL\_ROWS = 8;

// if any winning row has three values that are the same (and not EMPTY),

// then we have a winner

for(int row = 0; row < TOTAL\_ROWS; ++row)

{

if ( (board[WINNING\_ROWS[row][0]] != EMPTY) && (board[WINNING\_ROWS[row][0]] == board[WINNING\_ROWS[row][1]]) && (board[WINNING\_ROWS[row][1]] == board[WINNING\_ROWS[row][2]]) )

{

return board[WINNING\_ROWS[row][0]];

}

}

I loop through each possible way a player can win to see whether either player has three in a row. The if statement checks to see whether the three squares in question all contain the same value and are not EMPTY. If so, it means that the row has either three Xs or Os in it, and one side has won. The function then returns the piece in the first position of this winning row.

If neither player has won, I check for a tie game.

// since nobody has won, check for a tie (no empty squares left) if (count(board.begin(), board.end(), EMPTY) == 0)

return TIE;

If there are no empty squares on the board, then the game is a tie. I use the STL count() algorithm, which counts the number of times a given value appears in a group of container elements, to count the number of EMPTY elements in board. If the number is equal to 0, the function returns TIE.

Finally, if neither player has won and the game isn’t a tie, then there is no winner yet. Thus, the function returns NO\_ONE.

// since nobody has won and it isn’t a tie, the game ain’t over return NO\_ONE;

}

#### [The isLegal() Function](#_bookmark8)

This function receives a board and a move. It returns true if the move is a legal one on the board or false if the move is not legal. A legal move is represented by the number of an empty square.

Introducing the Tic-Tac-Toe Game 213

inline bool isLegal(int move, const vector<char>& board)

{

return (board[move] == EMPTY);

}

Again, notice that the vector that represents the board is passed through a constant reference. This means that the vector is passed efficiently; it is not copied. It also means that the vector is safeguarded against any change.

You can see that I inlined isLegal(). Modern compilers are quite good at optimizing on their own; however, since this function is just one line, it’s a good candidate for inlining.

#### [The humanMove() Function](#_bookmark8)

This next function receives a board and the human’s piece. It returns the square number for where the player wants to move. The function asks the player for the square number to which he wants to move until the response is a legal move. Then the function returns the move.

int humanMove(const vector<char>& board, char human)

{

int move = askNumber("Where will you move?", (board.size() - 1)); while (!isLegal(move, board))

{

cout << "\nThat square is already occupied, foolish human.\n"; move = askNumber("Where will you move?", (board.size() - 1));

}

cout << "Fine.. .\n";

return move;

}

Again, notice that the vector that represents the board is passed through a constant reference. This means that the vector is passed efficiently; it is not copied. It also means that the vector is safeguarded against any change.

#### [The computerMove() Function](#_bookmark8)

This function receives the board and the computer’s piece. It returns the computer’s move. The first thing to notice is that I do not pass the board by reference.

int computerMove(vector<char> board, char computer)

214 Chapter 6 n References: Tic-Tac-Toe

Instead, I choose to pass by value, even though it’s not as efficient as passing by reference. I pass by value because I need to work with and modify a copy of the board as I place pieces in empty squares to determine the best computer move. By working with a copy, I keep the original vector that represents the board safe.

Now on to the guts of the function. Okay, how do I program a bit of AI so the computer puts up a decent fight? Well, I came up with a basic three-step strategy for choosing a move.

1. If the computer can win on this move, make that move.
2. Otherwise, if the human can win on his next move, block him.
3. Otherwise, take the best remaining open square. The best square is the center. The next best squares are the corners, and then the rest of the squares.

The next section of the function implements Step 1.

{

unsigned int move = 0; bool found = false;

//if computer can win on next move, that’s the move to make while (!found && move < board.size())

{

if (isLegal(move, board))

{

board[move] = computer;

found = winner(board) == computer; board[move] = EMPTY;

}

if (!found)

{

++move;

}

}

I begin to loop through all of the possible moves, 0–8. For each move, I test to see whether the move is legal. If it is, I put the computer’s piece in the

Introducing the Tic-Tac-Toe Game 215

corresponding square and check to see whether the move gives the computer a win. Then I undo the move by making that square empty again. If the move didn’t result in a win for the computer, I go on to the next empty square. However, if the move did give the computer a win, then the loop ends—and I’ve found the move (found is true) that I want the computer to make (square number move) to win the game.

Next, I check to see if I need to go on to Step 2 of my AI strategy. If I haven’t found a move yet (found is false), then I check to see whether the human can win on his next move.

//otherwise, if human can win on next move, that’s the move to make if (!found)

{

move = 0;

char human = opponent(computer);

while (!found && move < board.size())

{

if (isLegal(move, board))

{

board[move] = human;

found = winner(board) == human; board[move] = EMPTY;

}

if (!found)

{

++move;

}

}

}

I begin to loop through all of the possible moves, 0–8. For each move, I test to see whether the move is legal. If it is, I put the human’s piece in the corresponding square and check to see whether the move gives the human a win. Then I undo the move by making that square empty again. If the move didn’t result in a win for the human, I go on to the next empty square. However, if the move did give the human a win, then the loop ends—and I’ve found the

216 Chapter 6 n References: Tic-Tac-Toe

move (found is true) that I want the computer to make (square number move) to block the human from winning on his next move.

Next, I check to see if I need to go on to Step 3 of my AI strategy. If I haven’t found a move yet (found is false), then I look through the list of best moves, in order of desirability, and take the first legal one.

//otherwise, moving to the best open square is the move to make if (!found)

{

move = 0;

unsigned int i = 0;

const int BEST\_MOVES[] = {4, 0, 2, 6, 8, 1, 3, 5, 7};

//pick best open square

while (!found && i < board.size())

{

move = BEST\_MOVES[i];

if (isLegal(move, board))

{

found = true;

}

++i;

}

}

At this point in the function, I’ve found the move I want the computer to make—whether that’s a move that gives the computer a win, blocks a winning move for the human, or is simply the best empty square available. So, I have the computer announce the move and return the corresponding square number.

cout << "I shall take square number " << move << endl; return move;

}

**Real World**

The Tic-Tac-Toe game considers only the next possible move. Programs that play serious games of strategy, such as chess, look far deeper into the consequences of individual moves and consider many levels of moves and countermoves. In fact, good computer chess programs can consider literally millions of board positions before making a move.

Summary 217

#### [The announceWinner() Function](#_bookmark8)

This function receives the winner of the game, the computer’s piece, and the human’s piece. The function announces the winner or declares a tie.

void announceWinner(char winner, char computer, char human)

{

if (winner == computer)

{

cout << winner << "’s won!\n";

cout << "As I predicted, human, I am triumphant once more -- proof\n"; cout << "that computers are superior to humans in all regards.\n";

}

else if (winner == human)

{

cout << winner << "’s won!\n";

cout << "No, no! It cannot be! Somehow you tricked me, human.\n"; cout << "But never again! I, the computer, so swear it!\n";

}

else

{

}

}

cout << "It’s a tie.\n";

cout << "You were most lucky, human, and somehow managed to tie me.\n"; cout << "Celebrate... for this is the best you will ever achieve.\n";

### [Summary](#_bookmark8)

In this chapter, you should have learned the following concepts:

n A reference is an alias; it’s another name for a variable.

n You create a reference using &—the referencing operator.

n A reference must be initialized when it’s defined.

n A reference can’t be changed to refer to a different variable.

n Whatever you do to a reference is done to the variable to which the reference refers.

n When you assign a reference to a variable, you create a new copy of the referenced value.

218 Chapter 6 n References: Tic-Tac-Toe

n When you pass a variable to a function by value, you pass a copy of the variable to the function.

n When you pass a variable to a function by reference, you pass access to the variable.

n Passing by reference can be more efficient than passing by value, especially when you are passing large objects.

n Passing a reference provides direct access to the argument variable passed to a function. As a result, the function can make changes to the argument variable.

n A constant reference can’t be used to change the value to which it refers.

You declare a constant reference by using the keyword const.

n You can’t assign a constant reference or a constant value to a non-constant reference.

n Passing a constant reference to a function protects the argument variable from being changed by that function.

n Changing the value of an argument variable passed to a function can lead to confusion, so game programmers consider passing a constant reference before passing a non-constant reference.

n Returning a reference can be more efficient than returning a copy of a value, especially when you are returning large objects.

n You can return a constant reference to an object so the object can’t be changed through the returned reference.

n A basic technique of game AI is to have the computer consider all of its legal moves and all of its opponent’s legal replies before deciding which move to take next.

### [Questions and Answers](#_bookmark8)

Q: Different programmers put the reference operator (&) in different places when declaring a reference. Where should I put it?

A: Three basic styles exist with regard to using the referencing operator. Some programmers opt for int& ref = var;, while others opt for int & ref = var;.

Questions and Answers 219

Still others opt for int &ref = var;. The computer is fine with all three. There are cases to be made for each style; however, the most important thing is to be consistent.

Q: Why can’t I initialize a non-constant reference with a constant value?

A: Because a non-constant reference allows you to change the value to which it refers.

Q: If I initialize a constant reference with a non-constant variable, can I change the value of the variable?

A: Not through the constant reference because when you declare a constant reference, you’re saying that the reference can’t be used to change the value to which it refers (even if that value can be changed by other means).

Q: How does passing a constant reference save overhead?

A: When you pass a large object to a function by value, your program makes a copy of the object. This can be an expensive operation depending on the size of the object. Passing a reference is like only passing access to the large object; it is an inexpensive operation.

Q: Can I make a reference to a reference?

A: Not exactly. You can assign one reference to another reference, but the new reference will simply refer to the value to which the original reference refers.

Q: What happens if I declare a reference without initializing it? A: Your compiler should complain because it’s illegal.

Q: Why should I avoid changing the value of a variable that I pass through a reference?

A: Because it could lead to confusion. It’s impossible to tell from only a function call whether a variable is being passed to change its value.

Q: Does that mean I should always pass a constant reference?

A: No. You can pass a non-constant reference to a function, but to most game programmers, this signals that you intend to change the argument variable’s value.

220 Chapter 6 n References: Tic-Tac-Toe

Q: If I don’t change the argument variables passed to functions, how should I get new information back to the calling code?

A: Use return values.

Q: Can I pass a literal through a non-constant reference?

A: No. If you try to pass a literal as a non-constant reference, you’ll generate a compile error.

Q: Is it impossible to pass a literal to a parameter that accepts a reference?  A: No, you can pass a literal as a constant reference.

Q: What happens when I return an object from a function?

A: Normally, your program creates a copy of the object and returns that. This can be an expensive operation, depending on the size of the object.

Q: Why return a reference?

A: It can be more efficient because returning a reference doesn’t involve copying an object.

Q: How can I lose the efficiency of returning a reference?

A: By assigning the returned reference to a variable. When you assign a reference to a variable, the computer must make a copy of the object to which the reference refers.

Q: What’s wrong with returning a reference to a local variable?

A: The local variable doesn’t exist once the function ends, which means that you’re returning a reference to a non-existent object, which is illegal.

### [Discussion Questions](#_bookmark8)

1. What are the advantages and disadvantages of passing an argument by value?
2. What are the advantages and disadvantages of passing a reference?
3. What are the advantages and disadvantages of passing a constant reference?

Exercises 221

1. What are the advantages and disadvantages of returning a reference?
2. Should game AI cheat in order to create a more worthy opponent?

### [Exercises](#_bookmark8)

1. Improve the Mad Lib game from Chapter 5 by using references to make the program more efficient.
2. What’s wrong with the following program?

int main()

{

int score; score = 1000;

float& rScore = score; return 0;

}

1. What’s wrong with the following function?

int& plusThree(int number)

{

int threeMore = number + 3; return threeMore;

}

*This page intentionally left blank*

|  |  |
| --- | --- |
|  |  |
| [chapter 7](#_bookmark8) | |

# [Pointers: Tic-Tac-Toe 2.0](#_bookmark8)

Pointers are a powerful part of C++. In some ways, they behave like iterators from the STL. Often, you can use them in place of references. But pointers offer functionality that no other part of the language can. In this chapter, you’ll learn the basic mechanics of pointers and get an idea of what they’re good for. Specifically, you’ll learn to:

n Declare and initialize pointers

n Dereference pointers

n Use constants and pointers

n Pass and return pointers

n Work with pointers and arrays

### [Understanding Pointer Basics](#_bookmark8)

Pointers have a reputation for being difficult to understand. In reality, the essence of pointers is quite simple—a *pointer* is a variable that can contain a memory address. Pointers give you the ability to work directly and efficiently with computer memory. Like iterators from the STL, they’re often used to access the contents of other variables. But before you can put pointers to good use in your game programs, you have to understand the basics of how they work.

223

224 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

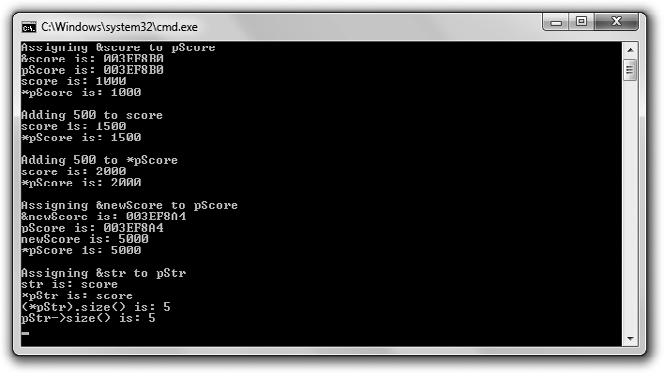
**Hi n t**

Computer memory is a lot like a neighborhood, but instead of houses in which people store their stuff, you have memory locations where you can store data. Just like a neighborhood where houses sit side by side, labeled with addresses, chunks of computer memory sit side by side, labeled with addresses. In a neighborhood, you can use a slip of paper with a street address on it to get to a particular house (and to the stuff stored inside it). In a computer, you can use a pointer with a memory address in it to get to a particular memory location (and to the stuff stored inside it).

#### [Introducing the Pointing Program](#_bookmark8)

The Pointing program demonstrates the mechanics of pointers. The program creates a variable for a score and then creates a pointer to store the address of that variable. The program shows that you can change the value of a variable directly, and the pointer will reflect the change. It also shows that you can change the value of a variable through a pointer. It then demonstrates that you can change a pointer to point to another variable entirely. Finally, the program shows that pointers can work just as easily with objects. Figure 7.1 illustrates the results of the program.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 7 folder; the filename is pointing.cpp.



**Figure 7.1**

The pointer pScore first points to the variable score and then to the variable newScore, while the pointer pStr points to the variable str.

Understanding Pointer Basics 225

// Pointing

// Demonstrates using pointers

#include <iostream> #include <string>

using namespace std; int main()

{

int\* pAPointer; //declare a pointer

int\* pScore = 0; //declare and initialize a pointer int score = 1000;

pScore = &score; //assign pointer pScore address of variable score

cout << "Assigning &score to pScore\n";

cout << "&score is: " << &score << "\n"; //address of score variable cout << "pScore is: " << pScore << "\n"; //address stored in pointer cout << "score is: " << score << "\n";

cout << "\*pScore is: " << \*pScore << "\n\n"; //value pointed to by pointer

cout << "Adding 500 to score\n"; score += 500;

cout << "score is: " << score << "\n";

cout << "\*pScore is: " << \*pScore << "\n\n";

cout << "Adding 500 to \*pScore\n";

\*pScore += 500;

cout << "score is: " << score << "\n";

cout << "\*pScore is: " << \*pScore << "\n\n";

cout << "Assigning &newScore to pScore\n"; int newScore = 5000;

pScore = &newScore;

cout << "&newScore is: " << &newScore << "\n"; cout << "pScore is: " << pScore << "\n";

cout << "newScore is: " << newScore << "\n"; cout << "\*pScore is: " << \*pScore << "\n\n";

226 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

cout << "Assigning &str to pStr\n"; string str = "score";

string\* pStr = &str; //pointer to string object cout << "str is: " << str << "\n";

cout << "\*pStr is: " << \*pStr << "\n";

cout << "(\*pStr).size() is: " << (\*pStr).size() << "\n"; cout << "pStr->size() is: " << pStr->size() << "\n";

return 0;

}

#### [Declaring Pointers](#_bookmark8)

With the first statement in main() I declare a pointer named pAPointer.

int\* pAPointer; //declare a pointer

Because pointers work in such a unique way, programmers often prefix pointer variable names with the letter “p” to remind them that the variable is indeed a pointer.

Just like an iterator, a pointer is declared to point to a specific type of value. pAPointer is a pointer to int, which means that it can only point to an int value. pAPointer can’t point to a float or a char, for example. Another way to say this is that pAPointer can only store the address of an int.

To declare a pointer of your own, begin with the type of object to which the pointer will point, followed by an asterisk, followed by the pointer name. When you declare a pointer, you can put whitespace on either side of the asterisk. So int\* pAPointer;, int \*pAPointer;, and int \* pAPointer; all declare a pointer named pAPointer.

**Trap**

When you declare a pointer, the asterisk only applies to the single variable name that immediately follows it. So the following statement declares pScore as a pointer to int and score as an int.

int\* pScore, score;

score is not a pointer! It’s a variable of type int. One way to make this clearer is to play with the whitespace and rewrite the statement as:

int \*pScore, score;

Understanding Pointer Basics 227

However, the clearest way to declare a pointer is to declare it in its own statement, as in the following lines.

int\* pScore; int score;

#### [Initializing Pointers](#_bookmark8)

As with other variables, you can initialize a pointer in the same statement you declare it. That’s what I do next with the following line, which assigns 0 to pScore.

int\* pScore = 0; //declare and initialize a pointer

Assigning 0 to a pointer has special meaning. Loosely translated, it means, “Point to nothing.” Programmers call a pointer with the value of zero a *null pointer*. You should always initialize a pointer with some value when you declare it, even if that value is zero.

**Hint**

Many programmers assign NULL to a pointer instead of 0 to make the pointer a null pointer. NULL is a constant defined in multiple library files, including iostream.

#### [Assigning Addresses to Pointers](#_bookmark8)

Because pointers store addresses of objects, you need a way to get addresses into the pointers. One way to do that is to get the memory address of an existing variable and assign it to a pointer. That’s what I do in the following line, which gets the address of the variable score and assigns it to pScore.

pScore = &score; //assign pointer address of variable score

I get the address of score by preceding the variable name with &, the *address of* operator. (Yes, you’ve seen the & symbol before, when it was used as the reference operator. However, in this context, the & symbol gets the address of an object.)

As a result of the preceding line of code, pScore contains the address of score. It’s as if pScore knows exactly where score is located in the computer’s memory. This means you can use pScore to get to score and manipulate the value stored in score. Figure 7.2 serves as a visual illustration of the relationship between pScore and score.

228 Chapter 7 n Pointers: Tic-Tac-Toe 2.0





**Figure 7.2**

The pointer pScore points to score, which stores the value 1000.

To prove that pScore contains the address of score, I display the address of the variable and the value of the pointer with the following lines.

cout << "&score is: " << &score << "\n"; //address of score variable cout << "pScore is: " << pScore << "\n"; //address stored in pointer

pScore contains 0x22ff5c, which is the address of score. (The specific addresses displayed by the Pointing program might be different on your system. The important thing is that the values for pScore and &score are the same.)

#### [Dereferencing Pointers](#_bookmark8)

Just as you dereference an iterator to access the object to which it refers, you dereference a pointer to access the object to which it points. You accomplish the dereferencing the same way—with \*, the dereference operator. I put the dereference operator to work with the following line, which displays 1000 because \*pScore accesses the value stored in score.

cout << "\*pScore is: " << \*pScore << "\n\n"; //value pointed to by pointer

Remember, \*pScore means, “the object to which pScore points.”

**Trap**

Don’t dereference a null pointer because it could lead to disastrous results.

Next, I add 500 to score with the following line.

score += 500;

When I send score to cout, 1500 is displayed, as you’d expect. When I send

\*pScore to cout, the contents of score are again sent to cout, and 1500 is displayed once more.

Understanding Pointer Basics 229

Next, I add 500 to the value to which pScore points with the following line.

\*pScore += 500;

Because pScore points to score, the preceding line of code adds 500 to score. Therefore, when I next send score to cout, 2000 is displayed. Then, when I send

\*pScore to cout.. .you guessed it, 2000 is displayed again.

**Trap**

Don’t change the value of a pointer when you want to change the value of the object to which the pointer points. For example, if I want to add 500 to the int that pScore points to, then the following line would be a big mistake.

pScore += 500;

The preceding code adds 500 to the address stored in pScore, not to the value to which pScore originally pointed. As a result, pScore now points to some address that might contain anything. Dereferencing a pointer like this can lead to disastrous results.

#### [Reassigning Pointers](#_bookmark8)

Unlike references, pointers can point to different objects at different times during the life of a program. Reassigning a pointer works like reassigning any other variable. Next, I reassign pScore with the following line.

pScore = &newScore;

As the result, pScore now points to newScore. To prove this, I display the address of newScore by sending &newScore to cout, followed by the address stored in pScore. Both statements display the same address. Then I send newScore and

\*pScore to cout. Both display 5000 because they both access the same chunk of memory that stores this value.

**Trap**

Don’t change the value to which a pointer points when you want to change the pointer itself. For example, if I want to change pScore to point to newScore, then the following line would be a big mistake.

\*pScore = newScore;

This code simply changes the value to which pScore currently points; it doesn’t change pScore itself. If newScore is equal to 5000, then the previous code is equivalent to \*pScore = 5000; and pScore still points to the same variable it pointed to before the assignment.

230 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

#### [Using Pointers to Objects](#_bookmark8)

So far, the Pointing program has worked only with values of a built-in type, int. But you can use pointers with objects just as easily. I demonstrate this next with the following lines, which create str, a string object equal to "score", and pStr, a pointer that points to that object.

string str = "score";

string\* pStr = &str; //pointer to string object

pStr is a pointer to string, meaning that it can point to any string object. Another way to say this is to say that pStr can store the address of any string object.

You can access an object through a pointer using the dereference operator. That’s what I do next with the following line.

cout << "\*pStr is: " << \*pStr << "\n";

By using the dereference operator with \*pStr, I send the object to which pStr

points (str) to cout. As a result, the text score is displayed.

You can call the member functions of an object through a pointer the same way you can call the member functions of an object through an iterator. One way to do this is by using the dereference operator and the member access operator, which is what I do next with the following lines.

cout << "(\*pStr).size() is: " << (\*pStr).size() << "\n";

The code (\*pStr).size() says, “Take the result of dereferencing pStr and call that object’s size() member function.” Because pStr refers to the string object equal to "score", the code returns 5.

**Hi n t**

Whenever you dereference a pointer to access a data member or member function, surround the dereferenced pointer with a pair of parentheses. This ensures that the dot operator will be applied to the object to which the pointer points.

Just as with iterators, you can use the -> operator with pointers for a more readable way to access object members. That’s what I demonstrate next with the following line.

cout << "pStr->size() is: " << pStr->size() << "\n";

Understanding Pointers and Constants 231

The preceding statement again displays the number of characters in the string object equal to "score"; however, I’m able to substitute pStr->size() for (\*pStr).size() this time, making the code more readable.

### [Understanding Pointers and Constants](#_bookmark9)

There are still some pointer mechanics you need to understand before you can start to use pointers effectively in your game programs. You can use the keyword const to restrict the way a pointer works. These restrictions can act as safeguards and can make your programming intentions clearer. Since pointers are quite versatile, restricting how a pointer can be used is in line with the programming mantra of asking for only what you need.

#### [Using a Constant Pointer](#_bookmark9)

As you’ve seen, pointers can point to different objects at different times in a program. However, by using the const keyword when you declare and initialize a pointer, you can restrict the pointer so it can only point to the object it was initialized to point to. A pointer like this is called a *constant pointer*. Another way to say this is to say that the address stored in a constant pointer can never change—it’s constant. Here’s an example of creating a constant pointer:

int score = 100;

int\* const pScore = &score; //a constant pointer

The preceding code creates a constant pointer, pScore, which points to score. You create a constant pointer by putting const right before the name of the pointer when you declare it.

Like all constants, you must initialize a constant pointer when you first declare it. The following line is illegal and will produce a big, fat compile error.

int\* const pScore; //illegal - - you must initialize a constant pointer

Because pScore is a constant pointer, it can’t ever point to any other memory location. The following code is also quite illegal.

pScore = &anotherScore; //illegal – pScore can’t point to a different object

Although you can’t change pScore itself, you can use pScore to change the value to which it points. The following line is completely legal.

\*pScore = 500;

232 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

Confused? Don’t be. It’s perfectly fine to use a constant pointer to change the value to which it points. Remember, the restriction on a constant pointer is that its value—the address that the pointer stores—can’t change.

The way a constant pointer works should remind you of something—a reference. Like a reference, a constant pointer can refer only to the object it was initialized to refer to.

**Hi n t**

Although you can use a constant pointer instead of a reference in your programs, you should stick with references when possible. References have a cleaner syntax than pointers and can make your code easier to read.

#### [Using a Pointer to a Constant](#_bookmark9)

As you’ve seen, you can use pointers to change the values to which they point. However, by using the const keyword when you declare a pointer, you can restrict a pointer so it can’t be used to change the value to which it points. A pointer like this is called a *pointer to a constant*. Here’s an example of declaring such a pointer:

const int\* pNumber; //a pointer to a constant

The preceding code declares a pointer to a constant, pNumber. You declare a pointer to a constant by putting const right before the type of value to which the pointer will point.

You assign an address to a pointer to a constant as you did before.

int lives = 3; pNumber = &lives;

However, you can’t use the pointer to change the value to which it points. The following line is illegal.

\*pNumber -= 1; //illegal - - can’t use pointer to a constant to change value

//that pointer points to

Although you can’t use a pointer to a constant to change the value to which it points, the pointer itself can change. This means that a pointer to a constant

Understanding Pointers and Constants 233

can point to different objects in a program. The following code is perfectly legal.

const int MAX\_LIVES = 5;

pNumber = &MAX\_LIVES; //pointer itself can change

#### [Using a Constant Pointer to a Constant](#_bookmark9)

A *constant pointer to a constant* combines the restrictions of a constant pointer and a pointer to a constant. This means that a constant pointer to a constant can only point to the object that it was initialized to point to. In addition, it can’t be used to change the value of the object to which it points. Here’s the declaration and initialization of such a pointer:

const int\* const pBONUS = &BONUS; //a constant pointer to a constant

The preceding code creates a constant pointer to a constant named pBONUS that points to the constant BONUS.

**Hint**

Like a pointer to a constant, a constant pointer to a constant can point to either a non-constant or constant value.

You can’t reassign a constant pointer to a constant. The following line is not legal.

pBONUS = &MAX\_LIVES; //illegal - - pBONUS can’t point to another object

You can’t use a constant pointer to a constant to change the value to which it points. This means that the following line is illegal.

\*pBONUS = MAX\_LIVES; //illegal - - can’t change value through pointer

In many ways, a constant pointer to a constant acts like a constant reference, which can only refer to the value it was initialized to refer to and which can’t be used to change that value.

**Hint**

Although you can use a constant pointer to a constant instead of a constant reference in your programs, you should stick with constant references when possible. References have a cleaner syntax than pointers and can make your code easier to read.

234 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

#### [Summarizing Constants and Pointers](#_bookmark9)

I’ve presented a lot of information on constants and pointers, so I want to provide a summary to help crystallize the new concepts. Here are three examples of the different ways in which you can use the keyword const when you are declaring pointers:

n int\* const p = &i;

n const int\* p;

n const int\* const p = &I;

The first example declares and initializes a constant pointer. A constant pointer can only point to the object to which it was initialized to point. The value—the memory address—stored in the pointer itself is constant and can’t change. A constant pointer can only point to a non-constant value; it can’t point to a constant.

The second example declares a pointer to a constant. A pointer to a constant can’t be used to change the value to which it points. A pointer to a constant can point to different objects during the life of a program. A pointer to a constant can point to a constant or non-constant value.

The third example declares a constant pointer to a constant. A constant pointer to a constant can only point to the value to which it was initialized to point. In addition, it can’t be used to change the value to which it points. A constant pointer to a constant can be initialized to point to a constant or a non-constant value.

### [Passing Pointers](#_bookmark9)

Even though references are the preferred way to pass arguments because of their cleaner syntax, you still might need to pass objects through pointers. For example, suppose you’re using a graphics engine that returns a pointer to a 3D object. If you want another function to use this object, you’ll probably want to pass the pointer to the object for efficiency. Therefore, it’s important to know how to pass pointers as well as references.

#### [Introducing the Swap Pointer Version Program](#_bookmark9)

The Swap Pointer Version program works just like the Swap program from Chapter 6, except that the Swap Pointer Version program uses pointers instead of references. The Swap Pointer Version program defines two variables—one

Passing Pointers 235

that holds my pitifully low score and another that holds your impressively high score. After displaying the scores, the program calls a function meant to swap the scores. Because only copies of the score values are sent to the function, the original variables are unaltered. Next, the program calls another swap function. This time, through the use of constant pointers, the original variables’ values are successfully exchanged (giving me the great big score and leaving you with the small one). Figure 7.3 shows the program in action.



**Figure 7.3**

Passing pointers allows a function to alter variables outside of the function’s scope.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 7 folder; the filename is swap\_pointer\_ver.cpp.

// Swap Pointer

// Demonstrates passing constant pointers to alter argument variables #include <iostream>

using namespace std;

void badSwap(int x, int y);

void goodSwap(int\* const pX, int\* const pY);

int main()

{

236 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

int myScore = 150; int yourScore = 1000;

cout << "Original values\n";

cout << "myScore: " << myScore << "\n";

cout << "yourScore: " << yourScore << "\n\n";

cout << "Calling badSwap()\n"; badSwap(myScore, yourScore);

cout << "myScore: " << myScore << "\n";

cout << "yourScore: " << yourScore << "\n\n";

cout << "Calling goodSwap()\n"; goodSwap(&myScore, &yourScore);

cout << "myScore: " << myScore << "\n"; cout << "yourScore: " << yourScore << "\n";

return 0;

}

void badSwap(int x, int y)

{

int temp = x; x = y;

y = temp;

}

void goodSwap(int\* const pX, int\* const pY)

{

//store value pointed to by pX in temp int temp = \*pX;

//store value pointed to by pY in address pointed to by pX

\*pX = \*pY;

//store value originally pointed to by pX in address pointed to by pY

\*pY = temp;

}

#### [Passing by Value](#_bookmark9)

After I declare and initialize myScore and yourScore, I send them to cout. As you’d expect, 150 and 1000 are displayed. Next I call badSwap(), which passes both

Passing Pointers 237

arguments by value. This means that when I call the function with the following line, copies of myScore and yourScore are sent to the parameters x and y.

badSwap(myScore, yourScore);

Specifically, x is assigned 150 and y is assigned 1000. As a result, nothing I do with x and y in badSwap() will have any effect on myScore and yourScore.

When badSwap() executes, x and y *do* exchange values—x becomes 1000 and y becomes 150. However, when the function ends, both x and y go out of scope. Control then returns to main(), in which myScore and yourScore haven’t changed. When I then send myScore and yourScore to cout, 150 and 1000 are displayed again. Sadly, I still have the tiny score and you still have the large one.

#### [Passing a Constant Pointer](#_bookmark9)

You’ve seen that it’s possible to give a function access to variables by passing references. It’s also possible to accomplish this using pointers. When you pass a pointer, you only pass the address of an object. This can be quite efficient, especially if you’re working with objects that occupy large chunks of memory. Passing a pointer is like e-mailing a friend the URL of a website instead of trying to send him the entire site.

Before you can pass a pointer to a function, you need to specify function parameters as pointers. That’s what I do in the goodSwap() header.

void goodSwap(int\* const pX, int\* const pY)

This means that pX and pY are constant pointers and will each accept a memory address. I made the parameters constant pointers because, although I plan to change the values they point to, I don’t plan to change the pointers themselves. Remember, this is just how references work. You can change the value to which a reference refers, but not the reference itself.

In main(), I pass the addresses of myScore and yourScore when I call goodSwap()

with the following line.

goodSwap(&myScore, &yourScore);

Notice that I send the addresses of the variables to goodSwap() by using the address of operator. When you pass an object to a pointer, you need to send the address of the object.

238 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

In goodSwap(), pX stores the address of myScore and pY stores the address of yourScore. Anything done to \*pX will be done to myScore; anything done to \*pY will be done to yourScore.

The first line of goodSwap() takes the value that pX points to and assigns it to temp.

int temp = \*pX;

Because pX points to myScore, temp becomes 150.

The next line assigns the value pointed to by pY to the object to which pX points.

\*pX = \*pY;

This statement copies the value stored in yourScore, 1000, and assigns it to the memory location of myScore. As a result, myScore becomes 1000.

The last statement in the function stores the value of temp, 150, in the address pointed to by pY.

\*pY = temp;

Because pY points to yourScore, yourScore becomes 150.

After the function ends, control returns to main(), where I send myScore and yourScore to cout. This time, 1000 and 150 are displayed. The variables have exchanged values. Success at last!

**Hi n t**

You can also pass a constant pointer to a constant. This works much like passing a constant reference, which is done to efficiently pass an object that you don’t need to change. I’ve adapted the Inventory Displayer program from Chapter 6, which demonstrates passing constant references, to pass a constant pointer to a constant. You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 7 folder; the filename is inventory\_displayer\_pointer\_ver.cpp.

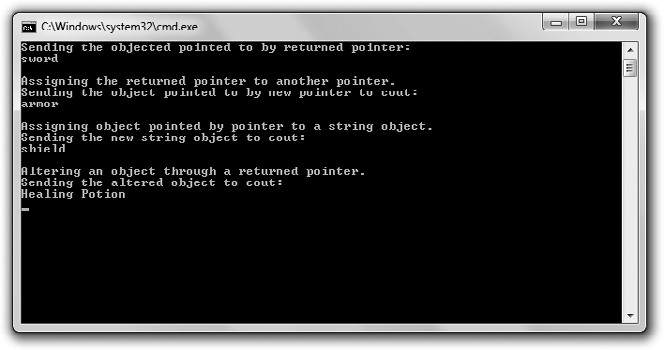
### [Returning Pointers](#_bookmark9)

Before references, the only option game programmers had for returning objects efficiently from functions was using pointers. And even though using references provides a cleaner syntax than using pointers, you might still need to return objects through pointers.

Returning Pointers 239

#### [Introducing the Inventory Pointer Program](#_bookmark9)

The Inventory Pointer program demonstrates returning pointers. Through returned pointers, the program displays and even alters the values of a vector that holds a hero’s inventory. Figure 7.4 shows the results of the program.



**Figure 7.4**

A function returns a pointer (not a string object) to each item in the hero’s inventory.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 7 folder; the filename is inventory\_pointer.cpp.

// Inventory Pointer

// Demonstrates returning a pointer

#include <iostream> #include <string> #include <vector>

using namespace std;

//returns a pointer to a string element

string\* ptrToElement(vector<string>\* const pVec, int i);

int main()

{

240 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

vector<string> inventory; inventory.push\_back("sword"); inventory.push\_back("armor"); inventory.push\_back("shield");

//displays string object that the returned pointer points to

cout << "Sending the objected pointed to by returned pointer:\n"; cout << \*(ptrToElement(&inventory, 0)) << "\n\n";

//assigns one pointer to another - - inexpensive assignment cout << "Assigning the returned pointer to another pointer.\n"; string\* pStr = ptrToElement(&inventory, 1);

cout << "Sending the object pointed to by new pointer to cout:\n"; cout << \*pStr << "\n\n";

//copies a string object - - expensive assignment

cout << "Assigning object pointed by pointer to a string object.\n"; string str = \*(ptrToElement(&inventory, 2));

cout << "Sending the new string object to cout:\n"; cout << str << "\n\n";

//altering the string object through a returned pointer cout << "Altering an object through a returned pointer.\n";

\*pStr = "Healing Potion";

cout << "Sending the altered object to cout:\n"; cout << inventory[1] << endl;

return 0;

}

string\* ptrToElement(vector<string>\* const pVec, int i)

{

//returns address of the string in position i of vector that pVec points to return &((\*pVec)[i]);

}

#### [Returning a Pointer](#_bookmark9)

Before you can return a pointer from a function, you must specify that you’re returning one. That’s what I do in the refToElement() header.

string\* ptrToElement(vector<string>\* const pVec, int i)

Returning Pointers 241

By starting the header with string\*, I’m saying that the function will return a pointer to a string object (and not a string object itself). To specify that a function returns a pointer to an object of a particular type, put an asterisk after the type name of the return type.

The body of the function ptrToElement() contains only one statement, which returns a pointer to the element at position i in the vector pointed to by pVec.

return &((\*pVec)[i]);

The return statement might look a little cryptic, so I’ll step through it. Whenever you come upon a complex expression, evaluate it like the computer does—by starting with the innermost part. I’ll start with (\*pVec)[i], which means the element in position i of the vector pointed to by pVec. By applying the address of operator (&) to the expression, it becomes the address of the element in position i of the vector pointed to by pVec.

**Trap**

Although returning a pointer can be an efficient way to send information back to a calling function, you have to be careful not to return a pointer that points to an out-of-scope object. For example, the following function returns a pointer that, if used, could crash the program.

string\* badPointer()

{

string local = "This string will cease to exist once the function ends."; string\* pLocal = &local;

return pLocal;

}

That’s because badPointer() returns a pointer to a string that no longer exists after the function ends. A pointer to a non-existent object is called a *dangling pointer*. Attempting to dereference a dangling pointer can lead to disastrous results. One way to avoid dangling pointers is to never return a pointer to a local variable.

#### [Using a Returned Pointer to Display a Value](#_bookmark9)

After I create inventory, a vector of items, I display a value with a returned pointer.

cout << \*(ptrToElement(&inventory, 0)) << "\n\n";

The preceding code calls ptrToElement(), which returns a pointer to inventory[0]. (Remember, ptrToElement() doesn’t return a copy of one of the elements of

242 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

inventory; it returns a pointer to one of them.) The line then sends the string

object pointed to by the pointer to cout. As a result, sword is displayed.

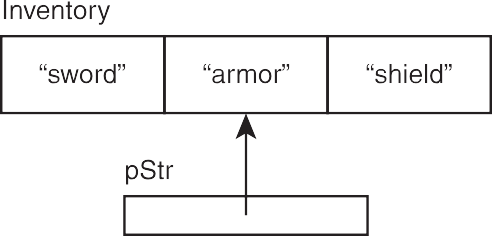
#### [Assigning a Returned Pointer to a Pointer](#_bookmark9)

Next I assign a returned pointer to another pointer with the following line.

string\* pStr = ptrToElement(&inventory, 1);

The call to prtToElement() returns a pointer to inventory[1]. The statement assigns that pointer to pStr. This is an efficient assignment because assigning a pointer to a pointer does not involve copying the string object.

To help you understand the results of this line of code, look at Figure 7.5, which shows a representation of pStr after the assignment. (Note that the figure is abstract because the vector inventory doesn’t contain the string literals "sword", "armor", and "shield"; instead, it contains string objects.)



**Figure 7.5**

pStr points to the element at position 1 of inventory.

Next I send \*pStr to cout and armor is displayed.

#### [Assigning to a Variable the Value Pointed to by a Returned Pointer](#_bookmark9)

Next I assign the value pointed to by a returned pointer to a variable.

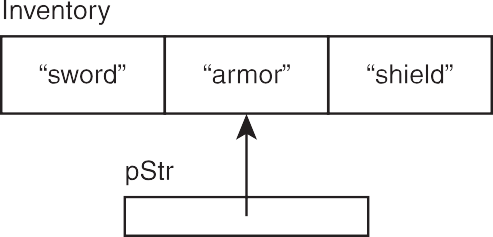
string str = \*(ptrToElement(&inventory, 2));

The call to prtToElement() returns a pointer to inventory[2]. However, the preceding statement doesn’t assign this pointer to str—it can’t because str is a string object. Instead, the computer quietly makes a copy of the string object to which the pointer points and assigns that object to str. To help drive this point home, check out Figure 7.6, which provides an abstract representation of the results of this assignment.

Returning Pointers 243







**Figure 7.6**

str is a new string object, totally independent from inventory.

An assignment like this one, where an object is copied, is more expensive than the assignment of one pointer to another. Sometimes the cost of copying an object is perfectly acceptable, but you should be aware of the extra overhead associated with this kind of assignment and avoid it when necessary.

#### [Altering an Object through a Returned Pointer](#_bookmark9)

You can also alter the object to which a returned pointer points. This means that I can change the hero’s inventory through pStr.

\*pStr = "Healing Potion";

Because pStr points to the element in position 1 of inventory, this code changes inventory[1] so it’s equal to "Healing Potion". To prove this, I display the element with the following line, which does indeed show Healing Potion.

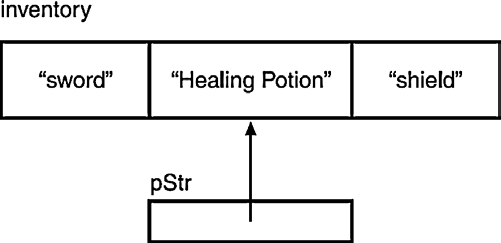
cout << inventory[1] << endl;

For an abstract representation, check out Figure 7.7, which shows the status of the variables after the assignment.

**Hint**

If you want to protect an object pointed to by a returned pointer, make sure to restrict the pointer. Return either a pointer to a constant or a constant pointer to a constant.

244 Chapter 7 n Pointers: Tic-Tac-Toe 2.0



**Figure 7.7**

inventory[1] is changed through the returned pointer stored in pStr.

### Understanding the Relationship between [Pointers and Arrays](#_bookmark9)

Pointers have an intimate relationship with arrays. In fact, an array name is really a constant pointer to the first element of the array. Because the elements of an array are stored in a contiguous block of memory, you can use the array name as a pointer for random access to elements. This relationship also has important implications for how you can pass and return arrays, as you’ll soon see.

#### [Introducing the Array Passer Program](#_bookmark9)

The Array Passer program creates an array of high scores and then displays them, using the array name as a constant pointer. Next, the program passes the array name as a constant pointer to a function that increases the scores. Finally, the program passes the array name to a function as a constant pointer to a constant to display the new high scores. Figure 7.8 shows the results of the program.

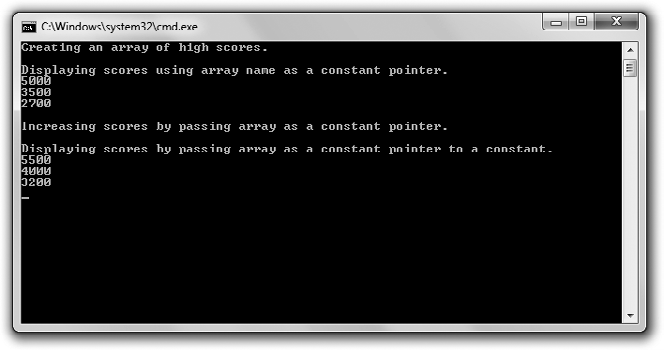
You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 7 folder; the filename is array\_passer.cpp.

//Array Passer

//Demonstrates relationship between pointers and arrays #include <iostream>

using namespace std;

Understanding the Relationship between Pointers and Arrays 245



**Figure 7.8**

Using an array name as a pointer, the high scores are displayed, altered, and passed to functions.

void increase(int\* const array, const int NUM\_ELEMENTS);

void display(const int\* const array, const int NUM\_ELEMENTS);

int main()

{

cout << "Creating an array of high scores.\n\n"; const int NUM\_SCORES = 3;

int highScores[NUM\_SCORES] = {5000, 3500, 2700};

cout << "Displaying scores using array name as a constant pointer.\n"; cout << \*highScores << endl;

cout << \*(highScores + 1) << endl; cout << \*(highScores + 2) << "\n\n";

cout << "Increasing scores by passing array as a constant pointer.\n\n"; increase(highScores, NUM\_SCORES);

cout << "Displaying scores by passing array as a constant pointer to a constant.\n";

display(highScores, NUM\_SCORES);

return 0;

}

246 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

void increase(int\* const array, const int NUM\_ELEMENTS)

{

for (int i = 0; i < NUM\_ELEMENTS; ++i)

{

array[i] += 500;

}

}

void display(const int\* const array, const int NUM\_ELEMENTS)

{

for (int i = 0; i < NUM\_ELEMENTS; ++i)

{

cout << array[i] << endl;

}

}

#### [Using an Array Name as a Constant Pointer](#_bookmark9)

Because an array name is a constant pointer to the first element of the array, you can dereference the name to get at the first element. That’s what I do after I create an array of high scores, called highScores.

cout << \*highScores << endl;

I dereference highScores to access the first element in the array and send it to

cout. As a result, 5000 is displayed.

You can randomly access array elements using an array name as a pointer through simple addition. All you have to do is add the position number of the element you want to access to the pointer before you dereference it. This is simpler than it sounds. For example, I next access the score at position 1 in highScores with the following line, which displays 3500.

cout << \*(highScores + 1) << endl;

In the preceding code, \*(highScores + 1) is equivalent to highScores[1]. Both return the element in position 1 of highScores.

Next, I access the score at position 2 in highScores with the following line, which displays 2700.

cout << \*(highScores + 2) << endl;

Understanding the Relationship between Pointers and Arrays 247

In the preceding code, \*(highScores + 2) is equivalent to highScores[2]. Both return the element in position 2 of highScores. In general, you can write arrayName[i] as \*(arrayName + i), where arrayName is the name of an array.

#### [Passing and Returning Arrays](#_bookmark9)

Because an array name is a constant pointer, you can use it to efficiently pass an array to a function. That’s what I do next with the following line, which passes to increase() a constant pointer to the first element of the array and the number of elements in the array.

increase(highScores, NUM\_SCORES);

**Hint**

When you pass an array to a function, it’s usually a good idea to also pass the number of elements in the array so the function can use this to avoid attempting to access an element that doesn’t exist.

As you can see from the function header of increase(), the array name is accepted as a constant pointer.

void increase(int\* const array, const int NUM\_ELEMENTS)

The function body adds 500 to each score.

for (int i = 0; i < NUM\_ELEMENTS; ++i)

{

array[i] += 500;

}

I treat array just like any array and use the subscripting operator to access each of its elements. Alternatively, I could have treated array as a pointer and substituted \*(array + i) += 500 for the expression array[i] += 500, but I opted for the more readable version.

After increase() ends, control returns to main(). To prove that increase() did in fact increase the high scores, I call a function to show the scores.

display(highScores, NUM\_SCORES);

The function display() also accepts highScore as a pointer. However, as you can see from the function’s header, the function accepts it as a constant pointer to a constant.

void display(const int\* const array, const int NUM\_ELEMENTS)

248 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

By passing the array in this way, I keep it safe from changes. Because all I want to do is display each element, it’s the perfect way to go.

Finally, the body of display() runs and all of the scores are listed, showing that they’ve each increased by 500.

**Hi n t**

You can pass a C-style string to a function, just like any other array. In addition, you can pass a string literal to a function as a constant pointer to a constant.

Because an array name is a pointer, you can return an array using the array name, just as you would any other pointer to an object.

### [Introducing the Tic-Tac-Toe 2.0 Game](#_bookmark9)

The project for this chapter is a modified version of the project from Chapter 6, the Tic-Tac-Toe game. From the player’s perspective, the Tic-Tac-Toe 2.0 game looks exactly the same as the original because the changes are under the hood— I’ve replaced all of the references with pointers. This means that objects such as the Tic-Tac-Toe board are passed as constant pointers instead of as references. This has other implications, including the fact that the address of a Tic-Tac-Toe board must be passed instead of the board itself.

You can download the code for the new version of the program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 7 folder; the filename is tic-tac-toe2.cpp. I won’t go over the code because most of it remains the same. But even though the number of changes isn’t great, the changes are significant. This is a good program to study because, although you should use references whenever you can, you should be equally comfortable with pointers.

### [Summary](#_bookmark9)

In this chapter, you should have learned the following concepts:

n Computer memory is organized in an ordered way, where each chunk of memory has its own unique address.

n A pointer is a variable that contains a memory address.

Summary 249

n In many ways, pointers act like iterators from the STL. For example, just as with iterators, you use pointers to indirectly access an object.

n To declare a pointer, you list a type, followed by an asterisk, followed by a name.

n Programmers often prefix pointer variable names with the letter “p” to remind them that the variable is indeed a pointer.

n Just like an iterator, a pointer is declared to refer to a value of a specific type. n It’s good programming practice to initialize a pointer when you declare it. n If you assign 0 to a pointer, the pointer is called a null pointer.

n To get the address of a variable, put the address of operator (&) before the variable name.

n When a pointer contains the address of an object, it’s said to point to the object.

n Unlike references, you can reassign pointers. That is, a pointer can point to different objects at different times during the life of a program.

n Just as with iterators, you dereference a pointer to access the object it points to with \*, the dereference operator.

n Just as with iterators, you can use the -> operator with pointers for a more readable way to access object data members and member functions.

n A constant pointer can only point to the object it was initialized to point to. You declare a constant pointer by putting the keyword const right before the pointer name, as in int\* const p = &i;.

n You can’t use a pointer to a constant to change the value to which it points. You declare a pointer to a constant by putting the keyword const before the type name, as in const int\* p;.

n A constant pointer to a constant can only point to the value it was initialized to point to, and it can’t be used to change that value. You declare a constant pointer to a constant by putting the keyword const before the type name and right before the pointer name, as in const int\* const p = &I;.

n You can pass pointers for efficiency or to provide direct access to an object.

250 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

n If you want to pass a pointer for efficiency, you should pass a pointer to a constant or a constant pointer to a constant so the object you’re passing access to can’t be changed through the pointer.

n A dangling pointer is a pointer to an invalid memory address. Dangling pointers are often caused by deleting an object to which a pointer pointed. Dereferencing such a pointer can lead to disastrous results.

n You can return a pointer from a function, but be careful not to return a dangling pointer.

### [Questions and Answers](#_bookmark9)

Q: How is a pointer different from the variable to which it points?

A: A pointer stores a memory address. If a pointer points to a variable, it stores the address of that variable.

Q: What good is it to store the address of a variable that already exists?

A: One big advantage of storing the address of an existing variable is that you can pass a pointer to the variable for efficiency instead of passing the variable by value.

Q: Does a pointer always have to point to an existing variable?

A: No. You can create a pointer that points to an unnamed chunk of computer memory as you need it. You’ll learn more about allocating memory in this dynamic fashion in Chapter 9, “Advanced Classes and Dynamic Memory: Game Lobby.”

Q: Why should I pass variables using references instead of pointers whenever possible?

A: Because of the sweet, syntactic sugar that references provide. Passing a reference or a pointer is an efficient way to provide access to objects, but pointers require extra syntax (like the dereference operator) to access the object itself.

Q: Why should I initialize a pointer when I declare it or soon thereafter?

A: Because dereferencing an uninitialized pointer can lead to disastrous results, including a program crash.

Questions and Answers 251

Q: What’s a dangling pointer?

A: A pointer that points to an invalid memory location, where any data could exist.

Q: What’s so dangerous about a dangling pointer?

A: Like using an uninitialized pointer, using a dangling pointer can lead to disastrous results, including a program crash.

Q: Why should I initialize a pointer to 0?

A: By initializing a pointer to 0, you create a null pointer, which is understood as a pointer to nothing.

Q: So then it’s safe to dereference a null pointer, right?

A: No! Although it’s good programming practice to assign 0 to a pointer that doesn’t point to an object, dereferencing a null pointer is as dangerous as dereferencing a dangling pointer.

Q: What will happen if I dereference a null pointer?

A: Just like dereferencing a dangling pointer or an uninitialized pointer, the results are unpredictable. Most likely, you’ll crash your program.

Q: What good are null pointers?

A: They’re often returned by functions as a sign of failure. For example, if a function is supposed to return a pointer to an object that represents the graphics screen, but that function couldn’t initialize the screen, it might return a null pointer.

Q: How does using the keyword const when declaring a pointer affect the pointer?

A: It depends on how you use it. Generally, you use const when you are declaring a pointer to restrict what the pointer can do.

Q: What kinds of restrictions can I impose on a pointer by declaring it with

const?

A: You can restrict a pointer so it can only point to the object it was initialized to point to, or you can restrict a pointer so it can’t change the value of the object it points to, or both.

252 Chapter 7 n Pointers: Tic-Tac-Toe 2.0

Q: Why would I want to restrict what a pointer can do?

A: For safety. For example, you might be working with an object that you know you don’t want to change.

Q: To what type of pointers can I assign a constant value?

A: A pointer to a constant or a constant pointer to a constant. Q: How can I safely return a pointer from a function?

A: One way is by returning a pointer to an object that you received from the calling function. This way, you’re returning a pointer to an object that exists back in the calling code. (In Chapter 9, you’ll discover another important way when you learn about dynamic memory.)

### [Discussion Questions](#_bookmark9)

1. What are the advantages and disadvantages of passing a pointer?
2. What kinds of situations call for a constant pointer?
3. What kinds of situations call for a pointer to a constant?
4. What kinds of situations call for a constant pointer to a constant?
5. What kinds of situations call for a non-constant pointer to a non-constant object?

### [Exercises](#_bookmark9)

1. Write a program with a pointer to a pointer to a string object. Use the pointer to the pointer to call the size() member function of the string object.
2. Rewrite the final project from Chapter 5, the Mad Lib game, so that no string objects are passed to the function that tells the story. Instead, the function should accept pointers to string objects.
3. Will the three memory addresses displayed by the following program all be the same? Explain what’s going on in the code.

Exercises 253

#include <iostream> using namespace std;

int main()

{

int a = 10; int& b = a; int\* c = &b;

cout << &a << endl; cout << &b << endl; cout << &(\*c) << endl;

return 0;

}

*This page intentionally left blank*

|  |  |
| --- | --- |
|  |  |
| [chapter 8](#_bookmark9) | |

# [Classes: Critter Caretaker](#_bookmark9)

*Object-oriented programming* (OOP) is a different way of thinking about programming. It’s a modern methodology that’s used in the creation of the vast majority of games (and other commercial software, too). In OOP, you define different types of objects with relationships to each other that allow the objects to interact. You’ve already worked with objects from types defined in libraries, but one of the key characteristics of OOP is the ability to make your own types from which you can create objects. In this chapter, you’ll see how to define your own types and create objects from them. Specifically, you’ll learn to:

n Create new types by defining classes

n Declare class data members and member functions

n Instantiate objects from classes

n Set member access levels

n Declare static data members and member functions

### [Defining New Types](#_bookmark9)

Whether you’re talking about alien spacecrafts, poisonous arrows, or angry mutant chickens, games are full of objects. Fortunately, C++ lets you represent game entities as software objects, complete with member functions and data members. These objects work just like the others you’ve already seen, such as

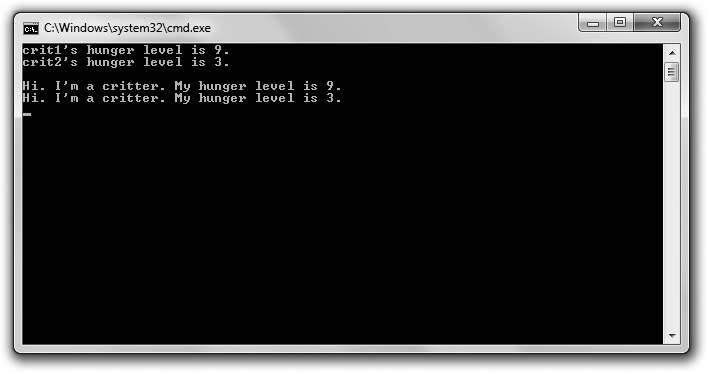
255

256 Chapter 8 n Classes: Critter Caretaker

string and vector objects. But to use a new kind of object (say, an angry mutant chicken object), you must first define a type for it.

#### [Introducing the Simple Critter Program](#_bookmark9)

The Simple Critter Program defines a brand-new type called Critter for creating virtual pet objects. The program uses this new type to create two Critter objects. Then, it gives each critter a hunger level. Finally, each critter offers a greeting and announces its hunger level to the world. Figure 8.1 shows the results of the program.



**Figure 8.1**

Each critter says hi and announces how hungry it is.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 8 folder; the filename is simple\_critter.cpp.

//Simple Critter

//Demonstrates creating a new type #include <iostream>

using namespace std;

Defining New Types 257

class Critter // class definition –– defines a new type, Critter

{

public:

int m\_Hunger; // data member

void Greet(); // member function prototype

};

void Critter::Greet() // member function definition

{

cout << "Hi. I’m a critter. My hunger level is " << m\_Hunger << ".\n";

}

int main()

{

Critter crit1;

Critter crit2;

crit1.m\_Hunger = 9;

cout << "crit1’s hunger level is " << crit1.m\_Hunger << ".\n";

crit2.m\_Hunger = 3;

cout << "crit2’s hunger level is " << crit2.m\_Hunger << ".\n\n";

crit1.Greet(); crit2.Greet();

return 0;

}

#### [Defining a Class](#_bookmark9)

To create a new type, you can define a *class*—code that groups data members and member functions. From a class, you create individual objects that have their own copies of each data member and access to all of the member functions. A class is like a blueprint. Just as a blueprint defines the structure of a building, a class defines the structure of an object. And just as a foreman can create many houses from the same blueprint, a game programmer can create many objects from the same class. Some real code will help solidify this theory. I begin a class definition in the Simple Critter program with

class Critter // class definition –– defines a new type, Critter

258 Chapter 8 n Classes: Critter Caretaker

for a class named Critter. To define a class, start with the keyword class, followed by the class name. By convention, class names begin with an uppercase letter. You surround the class body with curly braces and end it with a semicolon.

***Declaring Data Members***

In a class definition, you can declare class data members to represent object qualities. I give the critters just one quality, hunger. I see hunger as a range that could be represented by an integer, so I declare an int data member m\_Hunger.

int m\_Hunger; // data member

This means that every Critter object will have its own hunger level, represented by its own data member named m\_Hunger. Notice that I prefix the data member name with m\_. Some game programmers follow this naming convention so that data members are instantly recognizable.

***Declaring Member Functions***

In a class definition, you can also declare member functions to represent object abilities. I give a critter just one—the ability to greet the world and announce its hunger level—by declaring the member function Greet().

void Greet(); // member function prototype

This means that every Critter object will have the ability to say hi and announce its own hunger level through its member function, Greet(). By convention, member function names begin with an uppercase letter. At this point, I’ve only declared the member function Greet(). Don’t worry, though, I’ll define it outside of the class.

**Hi n t**

You might have noticed the keyword public in the class definition. You can ignore it for now. You’ll learn more about it a bit later in this chapter, in the section, “Specifying Public and Private Access Levels.”

#### [Defining Member Functions](#_bookmark9)

You can define member functions outside of a class definition. Outside of the Critter class definition, I define the Critter member function Greet(), which says hi and displays the critter’s hunger level.

Defining New Types 259

void Critter::Greet() // member function definition

{

cout << "Hi. I’m a critter. My hunger level is " << m\_Hunger << ".\n";

}

The definition looks like any other function definition you’ve seen, except for one thing—I prefix the function name with Critter::. When you define a member function outside of its class, you need to qualify it with the class name and scope resolution operator so the compiler knows that the definition belongs to the class.

In the member function, I send m\_Hunger to cout. This means that Greet() displays the value of m\_Hunger for the specific object through which the function is called. This simply means that the member function displays the critter’s hunger level. You can access the data members and member functions of an object in any member function simply by using the member’s name.

#### [Instantiating Objects](#_bookmark9)

When you create an object, you *instantiate* it from a class. In fact, specific objects are called *instances* of the class. In main(), I instantiate two instances of Critter.

Critter crit1;

Critter crit2;

As a result, I have two Critter objects—crit1 and crit2.

#### [Accessing Data Members](#_bookmark9)

It’s time to put these critters to work. Next, I give my first critter a hunger level.

crit1.m\_Hunger = 9;

The preceding code assigns 9 to crit1’s data member m\_Hunger. Just like when you are accessing an available member function of an object, you can access an available data member of an object using the member selection operator.

To prove that the assignment worked, I display the critter’s hunger level.

cout << "crit1’s hunger level is " << crit1.m\_Hunger << ".\n";

The preceding code displays crit1’s data member m\_Hunger and correctly shows 9. Just like when you are assigning a value to an available data member, you can

260 Chapter 8 n Classes: Critter Caretaker

get the value of an available data member through the member selection operator.

Next, I show that the same process works for another Critter object.

crit2.m\_Hunger = 3;

cout << "crit2’s hunger level is " << crit2.m\_Hunger << ".\n\n";

This time, I assign 3 to crit2’s data member m\_Hunger and display it.

So, crit1 and crit2 are both instances of Critter, and yet each exists independently and each has its own identity. Also, each has its own m\_Hunger data member with its own value.

#### [Calling Member Functions](#_bookmark9)

Next, I again put the critters through their paces. I get the first critter to give a greeting.

crit1.Greet();

The preceding code calls crit1’s Greet() member function. The function accesses the calling object’s m\_Hunger data member to form the greeting it displays. Because crit1’s m\_Hunger data member is 9, the function displays the text: Hi. I’m a critter. My hunger level is 9.

Finally, I get the second critter to speak up.

crit2.Greet();

The preceding code calls crit2’s Greet() member function. This function accesses the calling object’s m\_Hunger data member to form the greeting it displays. Because crit2’s m\_Hunger data member is 3, the function displays the text: Hi. I’m a critter. My hunger level is 3.

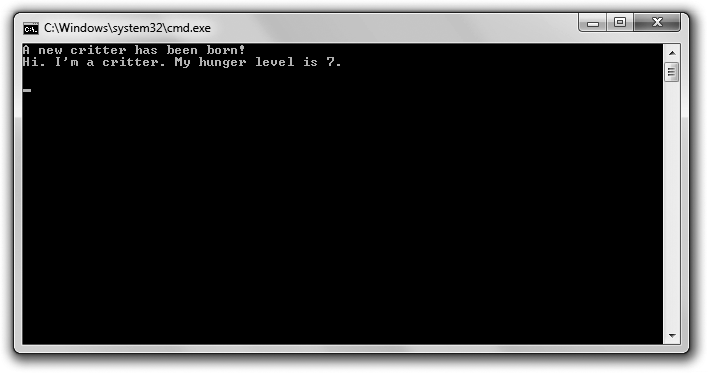
### [Using Constructors](#_bookmark9)

When you instantiate objects, you often want to do some initialization—usually assigning values to data members. Luckily, a class has a special member function known as a *constructor* that is automatically called every time a new object is instantiated. This is a big convenience because you can use a constructor to perform an initialization of the new object.

Using Constructors 261

#### [Introducing the Constructor Critter Program](#_bookmark9)

The Constructor Critter program demonstrates constructors. The program instantiates a new critter object, which automatically invokes its constructor. First, the constructor announces that a new critter has been born. Then, it assigns the value passed to it to the critter’s hunger level. Finally, the program calls the critter’s greeting member function, which displays the critter’s hunger level, proving that the constructor did in fact initialize the critter. Figure 8.2 shows the results of the program.



**Figure 8.2**

The Critter constructor initializes a new object’s hunger level automatically.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 8 folder; the filename is constructor\_critter.cpp.

//Constructor Critter

//Demonstrates constructors #include <iostream>

using namespace std;

class Critter

{

262 Chapter 8 n Classes: Critter Caretaker

public:

int m\_Hunger;

Critter(int hunger = 0); // constructor prototype void Greet();

};

Critter::Critter(int hunger) // constructor definition

{

cout << "A new critter has been born!" << endl; m\_Hunger = hunger;

}

void Critter::Greet()

{

cout << "Hi. I’m a critter. My hunger level is " << m\_Hunger << ".\n\n";

}

int main()

{

Critter crit(7); crit.Greet();

return 0;

}

#### [Declaring and Defining a Constructor](#_bookmark9)

I declare a constructor in Critter with the following code:

Critter(int hunger = 0); // constructor prototype

As you can see from the declaration, the constructor has no return type. It can’t— it’s illegal to specify a return type for a constructor. Also, you have no flexibility when naming a constructor. You have to give it the same name as the class itself.

**Hi n t**

A *default constructor* requires no arguments. If you don’t define a default constructor, the compiler defines a minimal one for you that simply calls the default constructors of any data members of the class. If you write your own constructor, then the compiler won’t provide a default constructor for you. It’s usually a good idea to have a default constructor, so you should make sure to supply your own when necessary. One way to accomplish this is to supply default arguments for all parameters in a constructor definition.

Using Constructors 263

I define the constructor outside of the class with the following code:

Critter::Critter(int hunger) // constructor definition

{

cout << "A new critter has been born!" << endl; m\_Hunger = hunger;

}

The constructor displays a message saying that a new critter has been born and initializes the object’s m\_Hunger data member with the argument value passed to the constructor. If no value is passed, then the constructor uses the default argument value of 0.

**Trick**

You can use *member initializers* as a shorthand way to assign values to data members in a constructor. To write a member initializer, start with a colon after the constructor’s parameter list. Then type the name of the data member you want to initialize, followed by the expression you want to assign to the data member, surrounded by parentheses. If you have multiple initializers, separate them with commas. This is much simpler than it sounds (and it’s really useful, too). Here’s an example that assigns hunger to m\_Hunger and boredom to m\_Boredom. Member initializers are especially useful when you have many data members to initialize.

Critter::Critter(int hunger = 0, int boredom = 0): m\_Hunger(hunger),

m\_Boredom(boredom)

{} // empty constructor body

#### [Calling a Constructor Automatically](#_bookmark9)

You don’t explicitly call a constructor; however, whenever you instantiate a new object, its constructor is automatically called. In main(), I put my constructor into action with the following code:

Critter crit(7);

When crit is instantiated, its constructor is automatically called and the message A new critter has been born! is displayed. Then, the constructor assigns 7 to the object’s m\_Hunger data member.

To prove that the constructor worked, back in main(), I call the object’s Greet() member function and sure enough, it displays Hi. I’m a critter. My hunger level is 7.

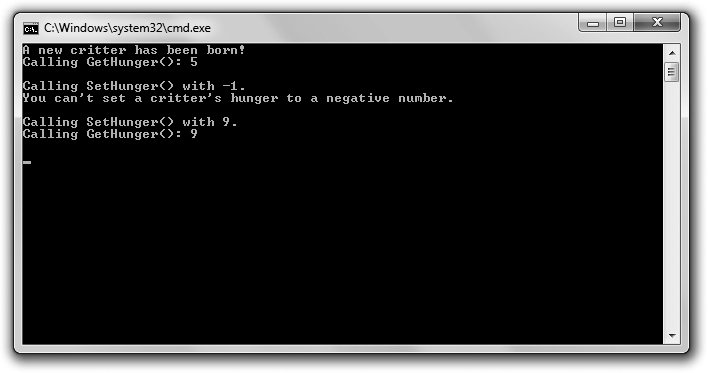
264 Chapter 8 n Classes: Critter Caretaker

### [Setting Member Access Levels](#_bookmark10)

Like functions, you should treat objects as encapsulated entities. This means that, in general, you should avoid directly altering or accessing an object’s data members. Instead, you should call an object’s member functions, allowing the object to maintain its own data members and ensure their integrity. Fortunately, you can enforce data member restrictions when you define a class by setting member access levels.

#### [Introducing the Private Critter Program](#_bookmark10)

The Private Critter program demonstrates class member access levels by declaring a class for critters that restricts direct access to an object’s data member for its hunger level. The class provides two member functions—one that allows access to the data member and one that allows changes to the data member. The program creates a new critter and indirectly accesses and changes the critter’s hunger level through these member functions. However, when the program attempts to change the critter’s hunger level to an illegal value, the member function that allows the changes catches the illegal value and doesn’t make the change. Finally, the program uses the hunger-level-setting member function with a legal value, which works like a charm. Figure 8.3 shows the results of the program.



**Figure 8.3**

By using a Critter object’s GetHunger() and SetHunger() member functions, the program indirectly accesses an object’s m\_Hunger data member.

Setting Member Access Levels 265

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 8 folder; the filename is private\_critter.cpp.

//Private Critter

//Demonstrates setting member access levels #include <iostream>

using namespace std;

class Critter

{

public: // begin public section Critter(int hunger = 0);

int GetHunger() const;

void SetHunger(int hunger);

private: // begin private section int m\_Hunger;

};

Critter::Critter(int hunger): m\_Hunger(hunger)

{

cout << "A new critter has been born!" << endl;

}

int Critter::GetHunger() const

{

return m\_Hunger;

}

void Critter::SetHunger(int hunger)

{

if (hunger < 0)

{

cout << "You can’t set a critter’s hunger to a negative number.\n\n";

}

266 Chapter 8 n Classes: Critter Caretaker

else

{

m\_Hunger = hunger;

}

}

int main()

{

Critter crit(5);

//cout << crit.m\_Hunger; //illegal, m\_Hunger is private! cout << "Calling GetHunger(): " << crit.GetHunger() << "\n\n";

cout << "Calling SetHunger() with -1.\n"; crit.SetHunger(-1);

cout << "Calling SetHunger() with 9.\n"; crit.SetHunger(9);

cout << "Calling GetHunger(): " << crit.GetHunger() << "\n\n";

return 0;

}

#### [Specifying Public and Private Access Levels](#_bookmark10)

Every class data member and member function has an access level, which determines from where in your program you can access it. So far, I’ve always specified class members to have public access levels using the keyword public. Again in Critter, I start a public section with the following line:

public: // begin public section

By using public:, I’m saying that any data member or member function that follows (until another access level specifier) will be public. This means that any part of the program can access them. Because I declare all of the member functions in this section, it means that any part of my code can call any member function through a Critter object.

Next, I specify a private section with the following line:

private: // begin private section

Setting Member Access Levels 267

By using private:, I’m saying that any data member or member function that follows (until another access level specifier) will be private. This means that only code in the Critter class can directly access it. Since I declare m\_Hunger in this section, it means that only the code in Critter can directly access an object’s m\_Hunger data member. Therefore, I can’t directly access an object’s m\_Hunger data member through the object in main() as I’ve done in previous programs. So the following line in main(), if uncommented, would be an illegal statement:

//cout << crit.m\_Hunger; //illegal, m\_Hunger is private!

Because m\_Hunger is private, I can’t access it from code that is not part of the Critter class. Again, only code that’s part of Critter can directly access the data member.

I’ve only shown you how to make data members private, but you can make member functions private, too. Also, you can repeat access modifiers. So if you want, you could have a private section, followed by a public section, followed by another private section in a class. Finally, member access is private by default. Until you specify an access modifier, any class members you declare will be private.

#### [Defining Accessor Member Functions](#_bookmark10)

An *accessor member function* allows indirect access to a data member. Because m\_Hunger is private, I wrote an accessor member function, GetHunger(), to return the value of the data member. (For now, you can ignore the keyword const.)

int Critter::GetHunger() const

{

return m\_Hunger;

}

I put the member function to work in main() with the following line:

cout << "Calling GetHunger(): " << crit.GetHunger() << "\n\n";

In the preceding code, crit.GetHunger() simply returns the value of crit’s m\_Hunger data member, which is 5.

268 Chapter 8 n Classes: Critter Caretaker

**Trick**

Just as you can with regular functions, you can inline member functions. One way to inline a member function is to define it right inside of the class definition, where you’d normally only declare the member function. If you include a member function definition in a class, then of course you don’t need to define it outside of the class.

An exception to this rule is that when you define a member function in a class definition using the keyword virtual, the member function is not automatically inlined. You’ll learn about virtual functions in Chapter 10, “Inheritance and Polymorphism: Blackjack.”

At this point, you might be wondering why you’d go to the trouble of making a data member private only to grant full access to it through accessor functions. The answer is that you don’t generally grant full access. For example, take a look at the accessor member function I defined for setting an object’s m\_Hunger data member, SetHunger():

void Critter::SetHunger(int hunger)

{

if (hunger < 0)

{

cout << "You can’t set a critter’s hunger to a negative number.\n\n";

}

else

{

m\_Hunger = hunger;

}

}

In this accessor member function, I first check to make sure that the value passed to the member function is greater than zero. If it’s not, it’s an illegal value and I display a message, leaving the data member unchanged. If the value is greater than zero, then I make the change. This way, SetHunger() protects the integrity of m\_Hunger, ensuring that it can’t be set to a negative number. Just as I’ve done here, most game programmers begin their accessor member function names with Get or Set.

#### [Defining Constant Member Functions](#_bookmark10)

A *constant member function* can’t modify a data member of its class or call a non-constant member function of its class. Why restrict what a member function can do? Again, it goes back to the tenet of asking only for what you

Using Static Data Members and Member Functions 269

need. If you don’t need to change any data members in a member function, then it’s a good idea to declare that member function to be constant. It protects you from accidentally altering a data member in the member function, and it makes your intentions clear to other programmers.

**Trap**

Okay, I lied a little. A constant member function can alter a static data member. You’ll learn about static data members a bit later in this chapter, in the “Declaring and Initializing Static Data Members” section. Also, if you qualify a data member with the mutable keyword, then even a constant member function can modify it. For now, though, don’t worry about either of these exceptions.

You can declare a constant member function by putting the keyword const at the end of the function header. That’s what I do in Critter with the following line, which declares GetHunger() to be a constant member function.

int GetHunger() const;

This means that GetHunger() can’t change the value of any non-static data member declared in the Critter class, nor can it call any non-constant Critter member function. I made GetHunger() constant because it only returns a value and doesn’t need to modify any data member. Generally, Get member functions can be defined as constant.

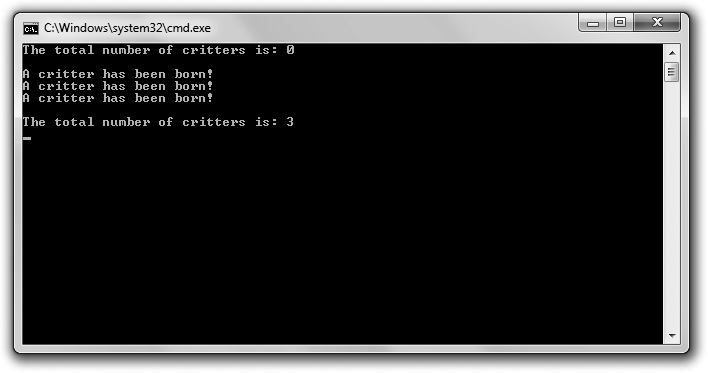
### Using Static Data Members [and Member Functions](#_bookmark10)

Objects are great because each instance stores its own set of data, giving it a unique identity. But what if you want to store some information about an entire class, such as the total number of instances that exist? You might want to do this if you’ve created a bunch of enemies and you want them to fight the player based on their total number. For example, if their total number is below a certain threshold, you might want the enemies to run away. You could store the total number of instances in each object, but that would be a waste of storage space. Plus, it would be cumbersome to update all of the objects as the total changes. Instead, what you really want is a way to store a single value for an entire class. You can do this with a static data member.

270 Chapter 8 n Classes: Critter Caretaker

#### [Introducing the Static Critter Program](#_bookmark10)

The Static Critter program declares a new kind of critter with a static data member that stores the total number of critters that have been created. It also defines a static member function that displays the total. Before the program instantiates any new critter objects, it displays the total number of critters by directly accessing the static data member that holds the total. Next, the program instantiates three new critters. Then it displays the total number of critters by calling a static member function that accesses the static data member. Figure 8.4 shows the results of the program.



**Figure 8.4**

The program stores the total number of Critter objects in the static data member s\_Total and accesses that data member in two different ways.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 8 folder; the filename is static\_critter.cpp.

//Static Critter

//Demonstrates static member variables and functions #include <iostream>

using namespace std;

Using Static Data Members and Member Functions 271

class Critter

{

public:

static int s\_Total; //static member variable declaration

//total number of Critter objects in existence

Critter(int hunger = 0);

static int GetTotal(); //static member function prototype

private:

int m\_Hunger;

};

int Critter::s\_Total = 0; //static member variable initialization Critter::Critter(int hunger):

m\_Hunger(hunger)

{

cout << "A critter has been born!" << endl;

++s\_Total;

}

int Critter::GetTotal() //static member function definition

{

return s\_Total;

}

int main()

{

cout << "The total number of critters is: "; cout << Critter::s\_Total << "\n\n";

Critter crit1, crit2, crit3;

cout << "\nThe total number of critters is: "; cout << Critter::GetTotal() << "\n";

return 0;

}

272 Chapter 8 n Classes: Critter Caretaker

#### [Declaring and Initializing Static Data Members](#_bookmark10)

A *static data member* is a single data member that exists for the entire class. In the class definition, I declare a static data member s\_Total to store the number of Critter objects that have been instantiated.

static int s\_Total; //static member variable declaration

You can declare your own static data members just like I did, by starting the declaration with the static keyword. I prefixed the variable name with s\_ so it would be instantly recognizable as a static data member.

Outside of the class definition, I initialize the static data member to 0.

int Critter::s\_Total = 0; //static member variable initialization

Notice that I qualified the data member name with Critter::. Outside of its class definition, you must qualify a static data member with its class name. After the previous line of code executes, there is a single value associated with the Critter class, stored in its static data member s\_Total with a value of 0.

**Hi n t**

You can declare a static variable in non-class functions, too. The static variable maintains its value between function calls.

#### [Accessing Static Data Members](#_bookmark10)

You can access a public static data member anywhere in your program. In main(), I access Critter::s\_Total with the following line, which displays 0, the value of the static data member and the total number of Critter objects that have been instantiated.

cout << Critter::s\_Total << "\n\n";

**Hi n t**

You can also access a static data member through any object of the class. Assuming that crit1 is a Critter object, I could display the total number of critters with the following line:

cout << crit1.s\_Total << "\n\n";

I also access this static data member in the Critter constructor with the following line, which increments s\_Total.

++s\_Total;

Using Static Data Members and Member Functions 273

This means that every time a new object is instantiated, s\_Total is incremented. Notice that I didn’t qualify s\_Total with Critter::. Just as with non-static data members, you don’t have to qualify a static data member with its class name inside its class.

Although I made my static data member public, you can make a static data member private—but then, like any other data member, you can only access it in a class member function.

#### [Declaring and Defining Static Member Functions](#_bookmark10)

A *static member function* exists for the entire class. I declare a static member function in Critter with the following line:

static int GetTotal(); //static member function prototype

You can declare your own static member function like I did, by starting the declaration with the keyword static. Static member functions are often written to work with static data members.

I define the static member function GetTotal() that returns the value of the static data member s\_Total.

int Critter::GetTotal() //static member function definition

{

return s\_Total;

}

A static member function definition is much like the non-static member function definitions you’ve seen so far. The major difference is that a static member function cannot access non-static data members. This is because a static member function exists for the entire class and is not associated with any particular instance of the class.

#### [Calling Static Member Functions](#_bookmark10)

After I instantiate three Critter objects in main(), I reveal the total number of critters again with the following line, which displays 3.

cout << Critter::GetTotal() << "\n\n";

To properly identify the static member function, I had to qualify it with Critter::. To call a static member function from outside of its class, you must qualify it with its class name.

274 Chapter 8 n Classes: Critter Caretaker

**Hi n t**

You can also access a static member function through any object of the class. Assuming that

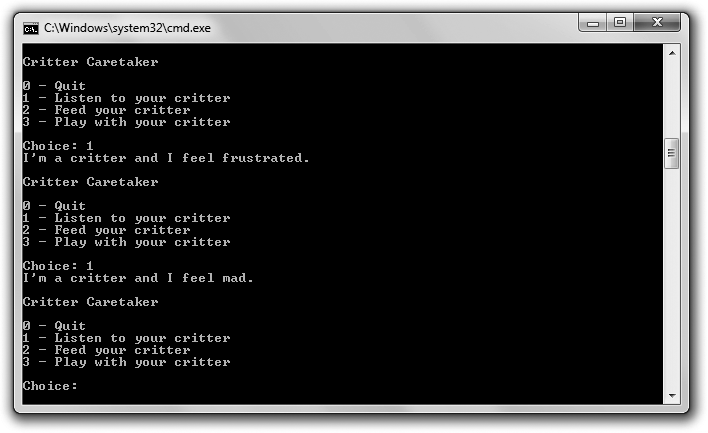
crit1 is a Critter object, I could display the total number of critters with the following line:

cout << crit1.GetTotal() << "\n\n";

Because static member functions exist for the entire class, you can call a static member function without any instances of the class in existence. And just as with private static data members, private static member functions can only be accessed by other member functions of the same class.

### [Introducing the Critter Caretaker Game](#_bookmark10)

The Critter Caretaker game puts the player in charge of his own virtual pet. The player is completely responsible for keeping the critter happy, which is no small task. He can feed and play with the critter to keep it in a good mood. He can also listen to the critter to learn how the critter is feeling, which can range from happy to mad. Figure 8.5 shows off the game.



**Figure 8.5**

If you fail to feed or entertain your critter, it will have a mood change for the worse. (But don’t worry— with the proper care, your critter can return to a sunny mood.)

Introducing the Critter Caretaker Game 275

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 8 folder; the filename is critter\_caretaker.cpp.

#### [Planning the Game](#_bookmark10)

The core of the game is the critter itself. Therefore, I first plan my Critter class. Because I want the critter to have independent hunger and boredom levels, I know that the class will have private data members for those.

n m\_Hunger

n m\_Boredom

The critter should also have a mood, directly based on its hunger and boredom levels. My first thought was to have a private data member, but a critter’s mood is really a calculated value based on its hunger and boredom. Instead, I decided to have a private member function that calculates a critter’s mood on the fly, based on its current hunger and boredom levels:

n GetMood()

Next, I think about public member functions. I want the critter to be able to tell the player how it’s feeling. I also want the player to be able to feed and play with the critter to reduce its hunger and boredom levels. I need three public member functions to accomplish each of these tasks.

n Talk() n Eat() n Play()

Finally, I want another member function that simulates the passage of time, to make the critter a little more hungry and bored:

n PassTime()

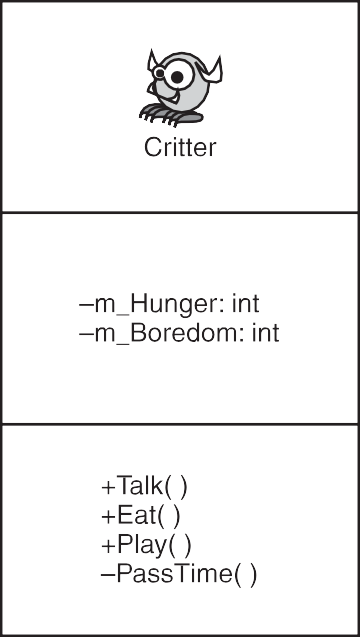
I see this member function as private because it will only be called by other member functions, such as Talk(), Eat(), or Play().

The class will also have a constructor to initialize data members. Take a look at Figure 8.6, which models the Critter class. I preface each data member and

276 Chapter 8 n Classes: Critter Caretaker

member function with a symbol to indicate its access level; I use + for public and

– for private.



**Figure 8.6**

Model for the Critter class.

#### [Planning the Pseudocode](#_bookmark10)

The rest of the program will be pretty simple. It’ll basically be a game loop that asks the player whether he wants to listen to, feed, or play with the critter, or quit the game. Here’s the pseudocode I came up with:

Create a critter

While the player doesn’t want to quit the game Present a menu of choices to the player

If the player wants to listen to the critter Make the critter talk

If the player wants to feed the critter Make the critter eat

If the player wants to play with the critter Make the critter play

Introducing the Critter Caretaker Game 277

#### [The Critter Class](#_bookmark10)

The Critter class is the blueprint for the object that represents the player’s critter. The class isn’t complicated, and most of it should look familiar, but it’s long enough that it makes sense to attack it in pieces.

***The Class Definition***

After some initial comments and statements, I begin the Critter class.

//Critter Caretaker

//Simulates caring for a virtual pet #include <iostream>

using namespace std;

class Critter

{

public:

Critter(int hunger = 0, int boredom = 0); void Talk();

void Eat(int food = 4); void Play(int fun = 4);

private:

int m\_Hunger; int m\_Boredom;

int GetMood() const;

void PassTime(int time = 1);

};

m\_Hunger is a private data member that represents the critter’s hunger level while m\_Boredom is a private data member that represents its boredom level. I’ll go through each member function in its own section.

***The Class Constructor***

The constructor takes two arguments, hunger and boredom. The arguments each have a default value of zero, which I specified in the constructor prototype back

278 Chapter 8 n Classes: Critter Caretaker

in the class definition. I use hunger to initialize m\_Hunger and boredom to initialize m\_Boredom.

Critter::Critter(int hunger, int boredom): m\_Hunger(hunger),

m\_Boredom(boredom)

{}

***The GetMood() Member Function***

Next, I define GetMood().

inline int Critter::GetMood() const

{

return (m\_Hunger + m\_Boredom);

}

The return value of this inlined member function represents a critter’s mood. As the sum of a critter’s hunger and boredom levels, a critter’s mood gets worse as the number increases. I made this member function private because it should only be invoked by another member function of the class. I made it constant since it won’t result in any changes to data members.

***The PassTime() Member Function***

PassTime() is a private member function that increases a critter’s hunger and boredom levels. It’s invoked at the end of each member function where the critter does something (eats, plays, or talks) to simulate the passage of time. I made this member function private because it should only be invoked by another member function of the class.

void Critter::PassTime(int time)

{

m\_Hunger += time; m\_Boredom += time;

}

You can pass the member function the amount of time that has passed; otherwise, time gets the default argument value of 1, which I specify in the member function prototype in the Critter class definition.

Introducing the Critter Caretaker Game 279

***The Talk() Member Function***

The Talk() member function announces the critter’s mood, which can be happy, okay, frustrated, or mad. Talk() calls GetMood() and, based on the return value, displays the appropriate message to indicate the critter’s mood. Finally, Talk() calls PassTime() to simulate the passage of time.

void Critter::Talk()

{

cout << "I’m a critter and I feel ";

int mood = GetMood(); if (mood > 15)

{

cout << "mad.\n";

}

else if (mood > 10)

{

cout << "frustrated.\n";

}

else if (mood > 5)

{

}

else

{

}

cout << "okay.\n";

cout << "happy.\n";

PassTime();

}

***The Eat() Member Function***

Eat() reduces a critter’s hunger level by the amount passed to the parameter food. If no value is passed, food gets the default argument value of 4. The critter’s hunger level is kept in check and is not allowed to go below zero. Finally, PassTime() is called to simulate the passage of time.

void Critter::Eat(int food)

{

cout << "Brruppp.\n"; m\_Hunger -= food;

280 Chapter 8 n Classes: Critter Caretaker

if (m\_Hunger < 0)

{

m\_Hunger = 0;

}

PassTime();

}

***The Play() Member Function***

Play() reduces a critter’s boredom level by the amount passed to the parameter fun. If no value is passed, fun gets the default argument value of 4. The critter’s boredom level is kept in check and is not allowed to go below zero. Finally, PassTime() is called to simulate the passage of time.

void Critter::Play(int fun)

{

cout << "Wheee!\n";

m\_Boredom -= fun; if (m\_Boredom < 0)

{

m\_Boredom = 0;

}

PassTime();

}

#### [The main() Function](#_bookmark10)

In main(), I instantiate a new Critter object. Because I don’t supply values for m\_Hunger or m\_Boredom, the data members start out at 0, and the critter begins life happy and content. Next, I create a menu system. If the player enters 0, the program ends. If the player enters 1, the program calls the object’s Talk() member function. If the player enters 2, the program calls the object’s Eat() member function. If the player enters 3, the program calls the object’s Play() member function. If the player enters anything else, he is told that the choice is invalid.

int main()

{

Critter crit; crit.Talk();

Summary 281

int choice; do

{

cout << "\nCritter Caretaker\n\n"; cout << "0 - Quit\n";

cout << "1 - Listen to your critter\n"; cout << "2 - Feed your critter\n";

cout << "3 - Play with your critter\n\n";

cout << "Choice: "; cin >> choice;

switch (choice)

{

case 0:

cout << "Good-bye.\n"; break;

case 1:

crit.Talk(); break;

case 2:

crit.Eat(); break;

case 3:

crit.Play(); break;

default:

cout << "\nSorry, but " << choice << " isn’t a valid choice.\n";

}

} while (choice != 0);

return 0;

}

### [Summary](#_bookmark10)

In this chapter, you should have learned the following concepts:

n Object-oriented programming (OOP) is a way of thinking about programming in which you define different types of objects with relationships that interact with each other.

282 Chapter 8 n Classes: Critter Caretaker

n You can create a new type by defining a class.

n A class is a blueprint for an object.

n In a class, you can declare data members and member functions.

n When you define a member function outside of a class definition, you need to qualify it with the class name and scope resolution operator (::).

n You can inline a member function by defining it directly in the class definition.

n You can access data members and member functions of objects through the member selection operator (.).

n Every class has a constructor—a special member function that’s automatically called every time a new object is instantiated. Constructors are often used to initialize data members.

n A default constructor requires no arguments. If you don’t provide a constructor definition in your class, the compiler will create a default constructor for you.

n Member initializers provide shorthand to assign values to data members in a constructor.

n You can set member access levels in a class by using the keywords public, private, and protected. (You’ll learn about protected in Chapter 9, “Advanced Classes and Dynamic Memory: Game Lobby.”)

n A public member can be accessed by any part of your code through an object.

n A private member can only be accessed by a member function of that class.

n An accessor member function allows indirect access to a data member.

n A static data member exists for the entire class.

n A static member function exists for the entire class.

n Some game programmers prefix private data member names with m\_ and static data member names with s\_ so that they’re instantly recognizable.

Questions and Answers 283

n A constant member function can’t modify non-static data members or call non-constant member functions of its class.

### [Questions and Answers](#_bookmark10)

Q: What is procedural programming?

A: A paradigm where tasks are broken down into a series of smaller tasks and implemented in manageable chunks of code, such as functions. In procedural programming, functions and data are separate.

Q: What is an object?

A: An entity that combines data and functions. Q: Why create objects?

A: Because the world—and especially game worlds—are full of objects. By creating your own types, you can represent objects and their relationships to other objects more directly and intuitively than you might be able to otherwise.

Q: What is object-oriented programming?

A: A paradigm where work is accomplished through objects. It allows pro- grammers to define their own types of objects. The objects usually have relationships to each other and can interact.

Q: Is C++ an object-oriented programming language or a procedural program- ming language?

A: C++ is a multi-paradigm programming language. It allows a game pro- grammer to write games in a procedural way or an object-oriented way—or through a combination of both (to name just a few options).

Q: Should I always try to write object-oriented game programs?

A: Although object-oriented programming is used in almost every commercial game on the market, you don’t have to write games using this paradigm. C++ lets you use one of several programming paradigms. In general, though, large game projects will almost surely benefit from an object-oriented approach.

284 Chapter 8 n Classes: Critter Caretaker

Q: Why not make all class members public?

A: Because it goes against the idea of encapsulation. Q: What is encapsulation?

A: The quality of being self-contained. In the world of OOP, encapsulation prevents client code from directly accessing the internals of an object. Instead, it encourages client code to use a defined interface to the object.

Q: What are the benefits of encapsulation?

A: In the world of OOP, encapsulation protects the integrity of an object. For example, you might have a spaceship object with a fuel data member. By preventing direct access to this data member, you can guarantee that it never becomes an illegal value (such as a negative number).

Q: Should I provide access to data members through accessor member functions?

A: Some game programmers say you should never provide access to data members through accessor member functions because even though this kind of access is indirect, it goes against the idea of encapsulation. Instead, they say you should write classes with member functions that provide the client with all of the functionality it could need, eliminating the client’s need to access a specific data member.

Q: What are mutable data members?

A: Data members that can be modified even by constant member functions. You create a mutable data member using the keyword mutable. You can also modify a mutable data member of a constant object.

Q: Why is it useful to have a default constructor?

A: Because there might be times when objects are automatically created without any argument values passed to a constructor—for example, when you create an array of objects.

Q: What is a structure?

A: A structure is very similar to a class. The only real difference is that the default access level for structures is public. You define a structure by using the keyword struct.

Exercises 285

Q: Why does C++ have both structures and classes? A: So that C++ retains backward compatibly with C. Q: When should I use structures?

A: Some game programmers use structures to group only data together, without functions (because that’s how C structures work). But it’s probably best to avoid structures whenever possible and use classes instead.

### [Discussion Questions](#_bookmark10)

1. What are the advantages and disadvantages of procedural programming?
2. What are the advantages and disadvantages of object-oriented programming?
3. Are accessor member functions a sign of poor class design? Explain.
4. How are constant member functions helpful to a game programmer?
5. When is it a good idea to calculate an object’s attribute on the fly rather than storing it as a data member?

### [Exercises](#_bookmark10)

1. Improve the Critter Caretaker program so that you can enter an unlisted menu choice that reveals the exact values of the critter’s hunger and boredom levels.
2. Change the Critter Caretaker program so that the critter is more expressive about its needs by hinting at how hungry and bored it is.
3. What design problem does the following program have?

#include <iostream> using namespace std;

class Critter

{

public:

int GetHunger() const {return m\_Hunger;}

286 Chapter 8 n Classes: Critter Caretaker

private:

int m\_Hunger;

};

int main()

{

Critter crit;

cout << crit.GetHunger() << endl; return 0;

}

|  |  |
| --- | --- |
|  |  |
| [chapter 9](#_bookmark10) | |

# [Advanced Classes and Dynamic Memory:](#_bookmark10)

[Game Lobby](#_bookmark10)

C++ gives a game programmer a high degree of control over the computer. One of the most fundamental abilities is direct control over memory. In this chapter, you’ll learn about *dynamic memory*—memory that you manage yourself. But with great power comes great responsibility, so you’ll also see the pitfalls of dynamic memory and how to avoid them. You’ll learn a few more things about classes, too. Specifically, you’ll learn to:

n Combine objects

n Use friend functions

n Overload operators

n Dynamically allocate and free memory

n Avoid memory leaks

n Produce deep copies of objects

### [Using Aggregation](#_bookmark10)

Game objects are often composed of other objects. For example, in a racing game, a drag racer could be seen as a single object composed of other individual objects, such as a body, four tires, and an engine. Other times, you might see an object as a collection of related objects. In a zookeeper simulation, you might see the zoo as a collection of an arbitrary number of animals. You can mimic these

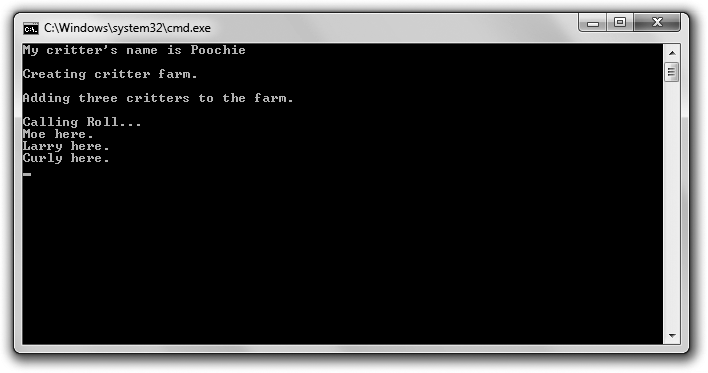
287

288 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

kinds of relationships among objects in OOP using *aggregation*—the combining of objects so that one is part of another. For example, you could write a Drag\_Racer class that has an engine data member that’s an Engine object. Or, you could write a Zoo class that has an animals data member that is a collection of Animal objects.

#### [Introducing the Critter Farm Program](#_bookmark10)

The Critter Farm program defines a new kind of critter with a name. After the program announces a new critter’s name, it creates a critter farm—a collection of critters. Finally, the program calls roll on the farm and each critter announces its name. Figure 9.1 shows the results of the program.



**Figure 9.1**

The critter farm is a collection of critters, each with a name.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 9 folder; the filename is critter\_farm.cpp.

//Critter Farm

//Demonstrates object containment

#include <iostream> #include <string> #include <vector>

Using Aggregation 289

using namespace std; class Critter

{

public:

Critter(const string& name = ""); string GetName() const;

private:

string m\_Name;

};

Critter::Critter(const string& name): m\_Name(name)

{}

inline string Critter::GetName() const

{

return m\_Name;

}

class Farm

{

public:

Farm(int spaces = 1);

void Add(const Critter& aCritter); void RollCall() const;

private:

vector<Critter> m\_Critters;

};

Farm::Farm(int spaces)

{

m\_Critters.reserve(spaces);

}

void Farm::Add(const Critter& aCritter)

{

m\_Critters.push\_back(aCritter);

}

290 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

void Farm::RollCall() const

{

for (vector<Critter>::const\_iterator iter = m\_Critters.begin(); iter != m\_Critters.end();

++iter)

{

cout << iter->GetName() << " here.\n";

}

}

int main()

{

Critter crit("Poochie");

cout << "My critter’s name is " << crit.GetName() << endl;

cout << "\nCreating critter farm.\n"; Farm myFarm(3);

cout << "\nAdding three critters to the farm.\n"; myFarm.Add(Critter("Moe")); myFarm.Add(Critter("Larry")); myFarm.Add(Critter("Curly"));

cout << "\nCalling Roll.. .\n"; myFarm.RollCall();

return 0;

}

#### [Using Object Data Members](#_bookmark10)

One way to use aggregation when you’re defining a class is to declare a data member that can hold another object. That’s what I did in Critter with the following line, which declares the data member m\_Name to hold a string object.

string m\_Name;

Generally, you use aggregation when an object has another object. In this case, a critter has a name. These kinds of relationships are called *has-a* relationships.

Using Aggregation 291

I put the declaration for the critter’s name to use when I instantiate a new object with:

Critter crit("Poochie");

which calls the Critter constructor:

Critter::Critter(const string& name): m\_Name(name)

{}

By passing the string literal "Poochie", the constructor is called and a string object for the name is instantiated, which the constructor assigns to m\_Name. A new critter named Poochie is born.

Next, I display the critter’s name with the following line:

cout << "My critter’s name is " << crit.GetName() << endl;

The code crit.GetName() returns a copy of the string object for the name of the critter, which is then sent to cout and displayed on the screen.

#### [Using Container Data Members](#_bookmark10)

You can also use containers as data members for your objects. That’s what I do when I define Farm. The single data member I declare for the class is simply a vector that holds Critter objects called m\_Critter.

vector<Critter> m\_Critters;

When I instantiate a new Farm object with:

Farm myFarm(3);

it calls the constructor:

Farm::Farm(int spaces)

{

m\_Critters.reserve(spaces);

}

which allocates memory for three Critter objects in the Farm object’s m\_Critter

vector.

Next, I add three critters to the farm by calling the Farm object’s Add() member function.

292 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

myFarm.Add(Critter("Moe")); myFarm.Add(Critter("Larry")); myFarm.Add(Critter("Curly"));

The following member function accepts a constant reference to a Critter object and adds a copy of the object to the m\_Critters vector.

void Farm::Add(const Critter& aCritter)

{

m\_Critters.push\_back(aCritter);

}

**Trap**

push\_back() adds a copy of an object to a vector—this means that I create an extra copy of each Critter object every time I call Add(). This is no big deal in the Critter Farm program, but if I were adding many large objects, it could become a performance issue. You can reduce this overhead by using, say, a vector of pointers to objects. You’ll see how to work with pointers to objects later in this chapter.

Finally, I take roll through the Farm object’s RollCall() member function.

myFarm.RollCall();

This iterates through the vector, calling each Critter object’s GetName() member function and getting each critter to speak up and say its name.

### Using Friend Functions [and Operator Overloading](#_bookmark10)

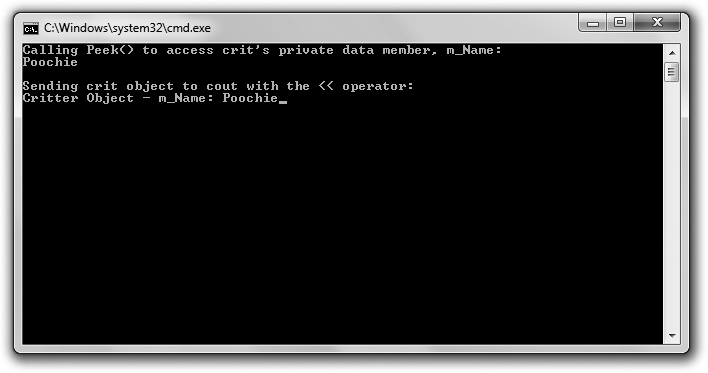
Friend functions and operator overloading are two advanced concepts related to classes. *Friend functions* have complete access to any member of a class. *Operator overloading* allows you to define new meanings for built-in operators as they relate to objects of your own classes. As you’ll see, you can use these two concepts together.

#### [Introducing the Friend Critter Program](#_bookmark10)

The Friend Critter program creates a critter object. It then uses a friend function, which is able to directly access the private data member that stores the critter’s name to display the critter’s name. Finally, the program displays the critter object by sending the object to the standard output. This is accomplished

Using Friend Functions and Operator Overloading 293

through a friend function and operator overloading. Figure 9.2 displays the results of the program.



**Figure 9.2**

The name of the critter is displayed through a friend function, and the critter object is displayed by sending it to the standard output.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 9 folder; the filename is friend\_critter.cpp.

//Friend Critter

//Demonstrates friend functions and operator overloading

#include <iostream> #include <string>

using namespace std; class Critter

{

//make following global functions friends of the Critter class friend void Peek(const Critter& aCritter);

friend ostream& operator<<(ostream& os, const Critter& aCritter);

294 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

public:

Critter(const string& name = "");

private:

string m\_Name;

};

Critter::Critter(const string& name): m\_Name(name)

{}

void Peek(const Critter& aCritter);

ostream& operator<<(ostream& os, const Critter& aCritter);

int main()

{

Critter crit("Poochie");

cout << "Calling Peek() to access crit’s private data member, m\_Name: \n"; Peek(crit);

cout << "\nSending crit object to cout with the << operator:\n"; cout << crit;

return 0;

}

//global friend function which can access all of a Critter object’s members void Peek(const Critter& aCritter)

{

cout << aCritter.m\_Name << endl;

}

//global friend function which can access all of Critter object’s members

//overloads the << operator so you can send a Critter object to cout ostream& operator<<(ostream& os, const Critter& aCritter)

{

os << "Critter Object - ";

os << "m\_Name: " << aCritter.m\_Name; return os;

}

Using Friend Functions and Operator Overloading 295

#### [Creating Friend Functions](#_bookmark10)

A friend function can access any member of a class of which it’s a friend. You specify that a function is a friend of a class by listing the function prototype preceded by the keyword friend inside the class definition. That’s what I do inside the Critter definition with the following line, which says that the global function Peek() is a friend of Critter.

friend void Peek(const Critter& aCritter);

This means Peek() can access any member of Critter even though it’s not a member function of the class. Peek() takes advantage of this relationship by accessing the private data member m\_Name to display the name of a critter passed to the function.

void Peek(const Critter& aCritter)

{

cout << aCritter.m\_Name << endl;

}

When I call Peek() in main() with the following line, the private data member

m\_Name of crit is displayed and Poochie appears on the screen.

Peek(crit);

#### [Overloading Operators](#_bookmark10)

Overloading operators might sound like something you want to avoid at all costs—as in, “Look out, that operator is overloaded and she’s about to blow!”— but it’s not. Operator overloading lets you give meaning to built-in operators used with new types that you define. For example, you could overload the \* operator so that when it is used with two 3D matrices (objects instantiated from some class that you’ve defined), the result is the multiplication of the matrices.

To overload an operator, define a function called operatorX, where X is the operator you want to overload. That’s what I do when I overload the << operator; I define a function named operator<<.

ostream& operator<<(ostream& os, const Critter& aCritter)

{

os << "Critter Object - ";

os << "m\_Name: " << aCritter.m\_Name; return os;

}

296 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

The function overloads the << operator so that when I send a Critter object with the << to cout, the data member m\_Name is displayed. Essentially, the function allows me to easily display Critter objects. The function can directly access the private data member m\_Name of a Critter object because I made the function a friend of the Critter class with the following line in Critter:

friend ostream& operator<<(ostream& os, const Critter& aCritter);

This means I can simply display a Critter object by sending it to cout with the

<< operator, which is what I do in main() with the following line, which displays the text Critter Object – m\_Name: Poochie.

cout << crit;

**Hi n t**

With all the tools and debugging options available to game programmers, sometimes simply displaying the values of variables is the best way to understand what’s happening in your programs. Overloading the << operator can help you do that.

This function works because cout is of the type ostream, which already overloads the << operator so that you can send built-in types to cout.

### [Dynamically Allocating Memory](#_bookmark10)

So far, whenever you’ve declared a variable, C++ has allocated the necessary memory for it. When the function that the variable was created in ended, C++ freed the memory. This memory, which is used for local variables, is called the *stack*. But there’s another kind of memory that persists independent of the functions in a program. You, the programmer, are in charge of allocating and freeing this memory, collectively called the *heap* (or *free store*).

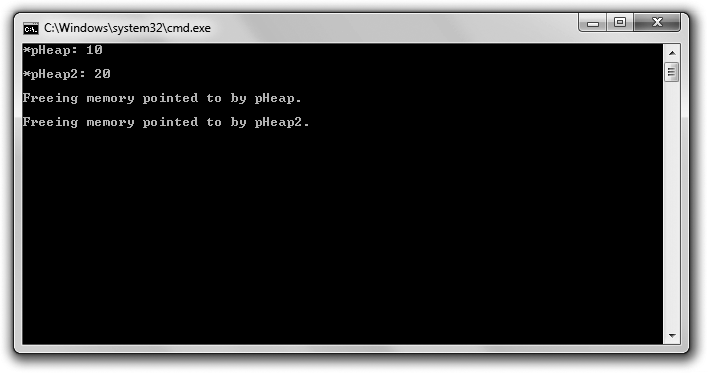
At this point, you might be thinking, “Why bother with another type of memory? The stack works just fine, thank you.” Using the dynamic memory of the heap offers great benefits that can be summed up in one word: efficiency. By using the heap, you can use only the amount of memory you need at any given time. If you have a game with a level that has 100 enemies, you can allocate the memory for the enemies at the beginning of the level and free the memory at the end. The heap also allows you to create an object in one function that you can access even after that function ends (without having to return a copy of the

Dynamically Allocating Memory 297

object). You might create a screen object in one function and return access to it. You’ll find that dynamic memory is an important tool in writing any significant game.

#### [Introducing the Heap Program](#_bookmark10)

The Heap program demonstrates dynamic memory. The program dynamically allocates memory on the heap for an integer variable, assigns it a value, and then displays it. Next, the program calls a function that dynamically allocates memory on the heap for another integer variable, assigns it a value, and returns a pointer to it. The program takes the returned pointer, uses it to display the value, and then frees the allocated memory on the heap. Finally, the program contains two functions that demonstrate the misuse of dynamic memory. I don’t call these functions, but I use them to illustrate what *not* to do with dynamic memory. Figure 9.3 shows the program.



**Figure 9.3**

The two int values are stored on the heap.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 9 folder; the filename is heap.cpp.

// Heap

// Demonstrates dynamically allocating memory

298 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

#include <iostream> using namespace std;

int\* intOnHeap(); //returns an int on the heap void leak1(); //creates a memory leak

void leak2(); //creates another memory leak

int main()

{

int\* pHeap = new int;

\*pHeap = 10;

cout << "\*pHeap: " << \*pHeap << "\n\n";

int\* pHeap2 = intOnHeap();

cout << "\*pHeap2: " << \*pHeap2 << "\n\n";

cout << "Freeing memory pointed to by pHeap.\n\n"; delete pHeap;

cout << "Freeing memory pointed to by pHeap2.\n\n"; delete pHeap2;

//get rid of dangling pointers pHeap = 0;

pHeap2 = 0;

return 0;

}

int\* intOnHeap()

{

int\* pTemp = new int(20); return pTemp;

}

void leak1()

{

int\* drip1 = new int(30);

}

Dynamically Allocating Memory 299

void leak2()

{

int\* drip2 = new int(50); drip2 = new int(100); delete drip2;

}

#### [Using the new Operator](#_bookmark10)

The new operator allocates memory on the heap and returns its address. You use new followed by the type of value you want to reserve space for. That’s what I do in the first line of main().

int\* pHeap = new int;

The new int part of the statement allocates enough memory on the heap for one int and returns the address on the heap for that chunk of memory. The other part of the statement, int\* pHeap, declares a local pointer, pHeap, which points to the newly allocated chunk of memory on the heap.

By using pHeap, I can manipulate the chunk of memory on the heap reserved for an integer. That’s what I do next; I assign 10 to the chunk of memory and then I display that value stored on the heap, using pHeap, as I would any other pointer to int. The only difference is that pHeap points to a piece of memory on the heap, not the stack.

**Hint**

You can initialize memory on the heap at the same time you allocate it by placing a value, surrounded by parentheses, after the type. This is even easier than it sounds. For example, the following line allocates a chunk of memory on the heap for an int variable and assigns 10 to it. The statement then assigns the address of that chunk of memory to pHeap.

int\* pHeap = new int(10);

One of the major advantages of memory on the heap is that it can persist beyond the function in which it was allocated, meaning that you can create an object on the heap in one function and return a pointer or reference to it. That’s what I demonstrate with the following line:

int\* pHeap2 = intOnHeap();

300 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

The statement calls the function intOnHeap(), which allocates a chunk of memory on the heap for an int and assigns 20 to it.

int\* intOnHeap()

{

int\* pTemp = new int(20); return pTemp;

}

Then, the function returns a pointer to this chunk of memory. Back in main(), the assignment statement assigns the address of the chunk of memory on the heap to pHeap2. Next, I use the returned pointer to display the value.

cout << "\*pHeap2: " << \*pHeap2 << "\n\n";

**Hi n t**

Up until now, if you wanted to return a value created in a function, you had to return a copy of the value. But by using dynamic memory, you can create an object on the heap in a function and return a pointer to the new object.

#### [Using the delete Operator](#_bookmark10)

Unlike storage for local variables on the stack, memory that you’ve allocated on the heap must be explicitly freed. When you’re finished with memory that you’ve allocated with new, you should free it with delete. That’s what I do with the following line, which frees the memory on the heap that stored 10.

delete pHeap;

That memory is returned to the heap for future use. The data that was stored in it is no longer available. Next, I free some more memory, which frees the memory on the heap that stored 20.

delete pHeap2;

That memory is returned to the heap for future use, and the data that was stored in it is no longer available. Notice that there’s no difference, as far as delete is concerned, regarding where in the program I allocated the memory on the heap that I’m deleting.

**Trick**

Because you need to free memory that you’ve allocated once you’re finished with it, a good rule of thumb is that every new should have a corresponding delete. In fact, some programmers write the delete statement just after writing the new statement whenever possible, so they don’t forget it.

Dynamically Allocating Memory 301

An important point to understand here is that the two previous statements free the memory on the heap, but they do not directly affect the local variables pHeap and pHeap2. This creates a potential problem because pHeap and pHeap2 now point to memory that has been returned to the heap, meaning that they point to memory that the computer can use in some other way at any given time. Pointers like this are called *dangling pointers* and they are quite dangerous. You should never attempt to dereference a dangling pointer. One way to deal with dangling pointers is to assign 0 to them, and that’s what I do with the following lines, which reassign both dangling pointers so they no longer point to some memory to which they should not point.

pHeap = 0;

pHeap2 = 0;

Another good way to deal with a dangling pointer is to assign a valid memory address to it.

**Trap**

Using delete on a dangling pointer can cause your program to crash. Be sure to set a dangling pointer to 0 or reassign it to point to a new, valid chunk of memory.

#### [Avoiding Memory Leaks](#_bookmark10)

One problem with allowing a programmer to allocate and free memory is that he might allocate memory and lose any way to get at it, thus losing any way to ever free it. When memory is lost like this, it’s called a *memory leak*. Given a large enough leak, a program might run out of memory and crash. As a game programmer, it’s your responsibility to avoid memory leaks.

I’ve written two functions in the Heap program that purposely create memory leaks in order to show you what *not* to do when using dynamic memory. The first function is leak1(), which simply allocates a chunk of memory on the heap for an int value and then ends.

void leak1()

{

int\* drip1 = new int(30);

}

302 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

If I were to call this function, memory would be lost forever. (Okay, it would be lost until the program ended.) The problem is that drip1, which is the only connection to the newly acquired chunk of memory on the heap, is a local variable and ceases to exist when the function leak1() ends. So, there’s no way to free the allocated memory. Take a look at Figure 9.4 for a visual representa- tion of how the leak occurs.





**Figure 9.4**

The memory that stores 30 can no longer be accessed to be freed, so it has leaked out of the system.

To avoid this memory leak, I could do one of two things. I could use delete to free the memory in leak1(), or I could return a copy of the pointer drip1. If I choose the second option, I have to make sure to free this memory in some other part of the program.

The second function that creates a memory leak is leak2().

void leak2()

{

int\* drip2 = new int(50); drip2 = new int(100); delete drip2;

}

This memory leak is a little more subtle, but there is still a leak. The first line in the function body, int\* drip2 = new int(50);, allocates a new piece of memory on the heap, assigns 50 to it, and has drip2 point to that piece memory. So far, so good. The second line, drip2 = new int(100);, points drip2 to a new piece of memory on the heap, which stores the 100. The problem is that the memory on the heap that stores 50 now has nothing pointing to it, so there is no way for the program to free that memory. As a result, that piece of memory has essentially leaked out of the system. Check out Figure 9.5 for a visual representation of how the leak occurs.

The last statement of the function, delete drip2;, frees the memory that stores

100, so this won’t be the source of another memory leak. But remember, the

Working with Data Members and the Heap 303



**Figure 9.5**

By changing drip2 so that it points to the memory that stores 100, the memory that stores 50 is no longer accessible and has leaked out of the system.

memory on the heap that stores 50 has still leaked out of the system. Also, I don’t worry about drip2, which technically has become a dangling pointer, because it will cease to exist when the function ends.

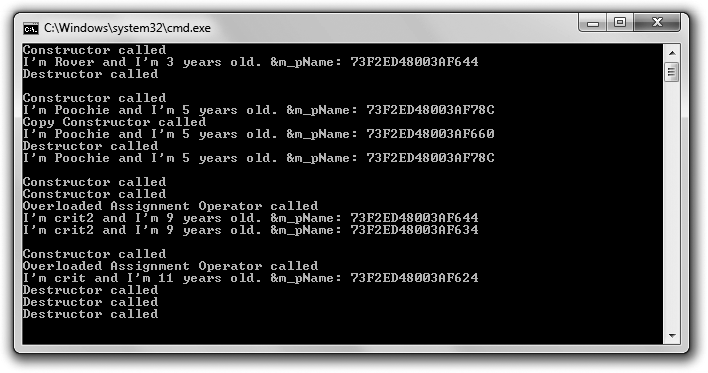
### [Working with Data Members and the Heap](#_bookmark10)

You’ve seen how you can use aggregation to declare data members that store objects, but you can also declare data members that are pointers to values on the heap. You might use a data member that points to a value on the heap for some of the same reasons you would use pointers in other situations. For example, you might want to declare a data member for a large 3D scene; however, you might only have access to the 3D scene through a pointer. Unfortunately, problems can arise when you use a data member that points to a value on the heap because of the way that some default object behaviors work. But you can avoid these issues by writing member functions to change these default behaviors.

#### [Introducing the Heap Data Member Program](#_bookmark10)

The Heap Data Member program defines a new type of critter with a data member that is a pointer, which points to an object stored on the heap. The class defines a few new member functions to handle situations in which an object is destroyed, copied, or assigned to another object. The program destroys, copies, and assigns objects to show that the objects behave as you’d expect, even with data members pointing to values on the heap. Figure 9.6 shows the results of the Heap Data Member program.

304 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby



**Figure 9.6**

Objects, each with a data member that points to a value on the heap, are instantiated, destroyed, and copied.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 9 folder; the filename is heap\_data\_member.cpp.

//Heap Data Member

//Demonstrates an object with a dynamically allocated data member

#include <iostream> #include <string>

using namespace std; class Critter

{

public:

Critter(const string& name = "", int age = 0);

~Critter(); //destructor prototype Critter(const Critter& c); //copy constructor prototype

Critter& Critter::operator=(const Critter& c); //overloaded assignment

op

void Greet() const;

Working with Data Members and the Heap 305

private:

string\* m\_pName; int m\_Age;

};

Critter::Critter(const string& name, int age)

{

cout << "Constructor called\n"; m\_pName = new string(name); m\_Age = age;

}

Critter::~Critter() //destructor definition

{

cout << "Destructor called\n"; delete m\_pName;

}

Critter::Critter(const Critter& c) //copy constructor definition

{

cout << "Copy Constructor called\n"; m\_pName = new string(\*(c.m\_pName)); m\_Age = c.m\_Age;

}

Critter& Critter::operator=(const Critter& c) //overloaded assignment op def

{

cout << "Overloaded Assignment Operator called\n"; if (this != &c)

{

delete m\_pName;

m\_pName = new string(\*(c.m\_pName)); m\_Age = c.m\_Age;

}

return \*this;

}

void Critter::Greet() const

{

306 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

cout << "I’m " << \*m\_pName << " and I’m " << m\_Age << " years old.\n"; cout << "&m\_pName: " << cout << &m\_pName << endl;

}

void testDestructor();

void testCopyConstructor(Critter aCopy); void testAssignmentOp();

int main()

{

testDestructor(); cout << endl;

Critter crit("Poochie", 5); crit.Greet(); testCopyConstructor(crit); crit.Greet();

cout << endl; testAssignmentOp();

return 0;

}

void testDestructor()

{

Critter toDestroy("Rover", 3); toDestroy.Greet();

}

void testCopyConstructor(Critter aCopy)

{

aCopy.Greet();

}

void testAssignmentOp()

{

Critter crit1("crit1", 7);

Critter crit2("crit2", 9); crit1 = crit2;

Working with Data Members and the Heap 307

crit1.Greet(); crit2.Greet(); cout << endl;

Critter crit3("crit", 11); crit3 = crit3; crit3.Greet();

}

#### [Declaring Data Members that Point to Values on the Heap](#_bookmark10)

To declare a data member that points to a value on the heap, you first need to declare a data member that’s a pointer. That’s just what I do in Critter with the following line, which declares m\_pName as a pointer to a string object.

string\* m\_pName;

In the class constructor, you can allocate memory on the heap, assign a value to the memory, and then point a pointer data member to the memory. That’s what I do in the constructor definition with the following line, which allocates memory for a string object, assigns name to it, and points m\_pName to that chunk of memory on the heap.

m\_pName = new string(name);

I also declare a data member that is not a pointer:

int m\_Age;

This data member gets its value in the constructor the way you’ve seen before, with a simple assignment statement:

m\_Age = age;

You’ll see how each of these data members is treated differently as Critter

objects are destroyed, copied, and assigned to each other.

Now, the first object with a data member on the heap is created when main() calls testDestructor(). The object, toDestroy, has an m\_pName data member that points to a string object equal to "Rover" that’s stored on the heap. Figure 9.7 provides a visual representation of the Critter object. Note that the image is abstract because the name of the critter is actually stored as a string object, not a string literal.

308 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby





**Figure 9.7**

A representation of a Critter object. The string object equal to "Rover" is stored on the heap.

#### [Declaring and Defining Destructors](#_bookmark11)

One problem that can occur when a data member of an object points to a value on the heap is a memory leak. That’s because when the object is deleted, the pointer to the heap value disappears along with it. If the heap value remains, it produces a memory leak. To avoid a memory leak, the object should clean up after itself before it is destroyed by deleting its associated heap value. Fortu- nately, there’s a member function, the *destructor*, that’s called just before an object is destroyed, which can be used to perform the necessary cleanup.

A default destructor, which is created for you by the compiler if you don’t write your own, doesn’t attempt to free any memory on the heap that a data member might point to. This behavior is usually fine for simple classes, but when you have a class with data members that point to values on the heap, you should write your own destructor so you can free the memory on the heap associated with an object before the object disappears, avoiding a memory leak. That’s what I do in the Critter class. First, inside the class definition, I declare the destructor. Notice that a destructor has the name of the class preceded by ~ (the tilde character) and does not have any parameters or return a value.

Critter::~Critter() //destructor definition

{

cout << "Destructor called\n"; delete m\_pName;

}

In main(), I put the destructor to the test when I call testDestructor(). The function creates a Critter object, toDestroy, and invokes its Greet() method, which displays I’m Rover and I’m 3 years old. &m\_pName: 73F2ED48003AF644. The message provides a way to see the values of the object’s m\_Age data member and the string pointed to by its m\_pName data member. But it also displays the address of the string on the heap stored in the pointer m\_pName. The important thing to

Working with Data Members and the Heap 309

note is that after the Greet() message is displayed, the function ends and toDestroy is ready to be destroyed. Fortunately, toDestroy’s destructor is automatically called just before this happens. The destructor displays Destructor called and deletes the string object equal to "Rover" that’s on the heap, cleaning up after itself and leaking no memory. The destructor doesn’t do anything with the m\_Age data member. That’s perfectly fine since m\_Age isn’t on the heap, but part of toDestroy and will be properly disposed of right along with the rest of the Critter object.

**Hint**

When you have a class that allocates memory on the heap, you should write a destructor that cleans up and frees that memory.

#### [Declaring and Defining Copy Constructors](#_bookmark11)

Sometimes an object is copied automatically for you. This occurs when an object is:

n Passed by value to a function

n Returned from a function

n Initialized to another object through an initializer

n Provided as a single argument to the object’s constructor

The copying is done by a special member function called the *copy constructor*. Like constructors and destructors, a default copy constructor is supplied for you if you don’t write one of your own. The default copy constructor simply copies the value of each data member to data members of the same name in the new object—a *member-wise copy*.

For simple classes, the default copy constructor is usually fine. However, when you have a class with a data member that points to a value on the heap, you should consider writing your own copy constructor. Why? Imagine a Critter object that has a data member that’s a pointer to a string object on the heap. With only a default copy constructor, the automatic copying of the object would result in a new object that points to the same single string on the heap because the pointer of the new object would simply get a copy of the address stored in

310 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

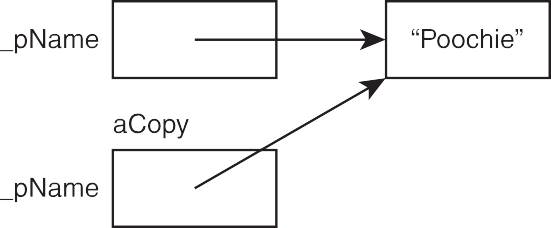
the pointer of the original object. This member-wise copying produces a *shallow copy*, in which the pointer data members of the copy point to the same chunks of memory as the pointer data members in the original object.

Let me give you a specific example. If I hadn’t written my own copy constructor in the Heap Data Member program, when I passed a Critter object by value with the following function call, the program would have automatically made a shallow copy of crit called aCopy that existed in testCopyConstructor().

testCopyConstructor(crit);

aCopy’s m\_pName data member would point to the exact same string object on the heap as crit’s m\_pName data member does. Figure 9.8 shows you what I mean. Note that the image is abstract since the name of the critter is actually stored as a string object, not a string literal.





**Figure 9.8**

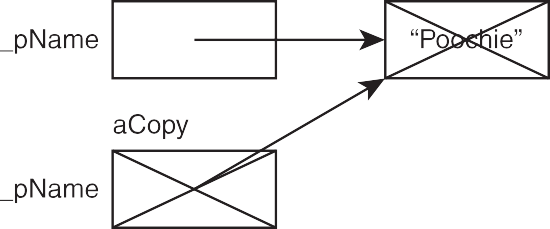
If a shallow copy of crit were made, both aCopy and crit would have a data member that points to the same chunk of memory on the heap.

Why is this a problem? Once testCopyConstructor() ends, aCopy’s destructor is called, freeing the memory on the heap pointed to by aCopy’s m\_pName data member. Because of this, crit’s m\_pName data member would point to memory that has been freed, which would mean that crit’s m\_pName data member would be a dangling pointer! Figure 9.9 provides you with a visual representation of this. Note that the image is abstract since the name of the critter is actually stored as a string object, not a string literal.

What you really need is a copy constructor that produces a new object with its own chunk of memory on the heap for each data member that points to a heap object—a *deep copy*. That’s what I do when I define a copy constructor for the

Working with Data Members and the Heap 311





**Figure 9.9**

If the shallow copy of the Critter object were destroyed, the memory on the heap that it shared with the original object would be freed. As a result, the original object would have a dangling pointer.

class, which replaces the default one provided by the compiler. First, inside the class definition, I declare the copy constructor:

Critter(const Critter& c); //copy constructor prototype

Next, outside the class definition, I define the copy constructor:

Critter::Critter(const Critter& c) //copy constructor definition

{

cout << "Copy Constructor called\n"; m\_pName = new string(\*(c.m\_pName)); m\_Age = c.m\_Age;

}

Just like this one, a copy constructor must have the same name as the class. It returns no value, but accepts a reference to an object of the class—the object that needs to be copied. The reference should be made a constant reference to protect the original object from being changed during the copy process.

The job of a copy constructor is to copy any data members from the original object to the copy object. If a data member of the original object is a pointer to a value on the heap, the copy constructor should request memory from the heap, copy the original heap value to this new chunk of memory, and then point the appropriate copy object data member to this new memory.

When I call testCopyConstructor() by passing crit to the function by value, the copy constructor I wrote is automatically called. You can tell this because the text Copy Constructor called. appears on the screen. My copy constructor creates a new Critter object (the copy) and accepts a reference to the original in c.

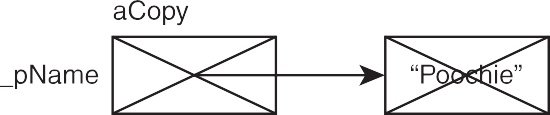
312 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

With the line m\_pName = new string(\*(c.m\_pName));, my copy constructor allocates a new chunk of memory on the heap, gets a copy of the string pointed to by the original object, copies it to the new memory, and points the m\_pName data member of the copy to this memory. The next line, m\_Age = c.m\_Age; simply copies the value of the original’s m\_Age to the copy’s m\_Age data member. As a result, a deep copy of crit is made, and that’s what gets used in testCopyConstructor() as aCopy.

You can see that the copy constructor worked when I called aCopy’s Greet() member function. In my sample run, the member function displayed a message, part of which was I’m Poochie and I’m 5 years old. This part of the message shows that aCopy correctly got a copy of the values of the data members from the object crit. The second part of the message, &m\_pName: 73F2ED48003AF660, shows that the string object pointed to by the data member m\_pName of aCopy is stored in a different chunk of memory than the string pointed to by the data member m\_pName of crit, which is stored at memory location 73F2ED48003AF78C, proving that a deep copy was made. Remember that the memory addresses displayed in my sample run may be different from the ones displayed when the program is run again. However, the key here is that the addresses stored in crit’s m\_pName and aCopy’s m\_pName are different from each other.

When testCopyConstructor() ends, the copy of the Critter object used in the function, stored in the variable aCopy, is destroyed. The destructor frees the chunk of memory on the heap associated with the copy, leaving the original Critter object, crit, created in main(), unaffected. Figure 9.10 shows the results. Note that the image is abstract since the name of the critter is actually stored as a string object, not a string literal.





**Figure 9.10**

With a proper copy constructor, the original and the copy each point to their own chunk of memory on the heap. Then, when the copy is destroyed, the original is unaffected.

Working with Data Members and the Heap 313

**Hint**

When you have a class with data members that point to memory on the heap, you should consider writing a copy constructor that allocates memory for a new object and creates a deep copy.

#### [Overloading the Assignment Operator](#_bookmark11)

When both sides of an assignment statement are objects of the same class, the class’ assignment operator member function is called. Like a default copy constructor, a default assignment operator member function is supplied for you if you don’t write one of your own. Also like the default copy constructor, the default assignment operator provides only member-wise duplication.

For simple classes, the default assignment operator is usually fine. However, when you have a class with a data member that points to a value on the heap, you should consider writing an overloaded assignment operator of your own. If you don’t, you’ll end up with shallow copies of objects when you assign one object to another. To avoid this problem, I overloaded the assignment operator for Critter. First, inside the class definition, I write the declaration:

Critter& Critter::operator=(const Critter& c); //overloaded assignment

op

Next, outside the class definition, I write the member function definition:

Critter& Critter::operator=(const Critter& c) //overloaded assignment op def

{

cout << "Overloaded Assignment Operator called\n"; if (this != &c)

{

delete m\_pName;

m\_pName = new string(\*(c.m\_pName)); m\_Age = c.m\_Age;

}

return \*this;

}

Notice that the member function returns a reference to a Critter object. For robust assignment operation, return a reference from the overloaded assignment operator member function.

314 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

In main(), I call a function that tests the overloaded assignment operator for this class.

testAssignmentOp();

The testAssignmentOp() creates two objects and assigns one to the other.

Critter crit1("crit1", 7);

Critter crit2("crit2", 9); crit1 = crit2;

The preceding assignment statement, crit1 = crit2;, calls the assignment operator member function—operator=()—for crit1. In the operator=() func- tion, c is a constant reference to crit2. The goal of the member function is to assign the values of all of the data members of crit2 to crit1 while making sure each Critter object has its own chunks of memory on the heap for any pointer data members.

After operator=() displays a message that the overloaded assignment operator has been called, it uses the this pointer. What’s the this pointer? It’s a pointer that all non-static member functions automatically have, which points to the object that was used to call the function. In this case, this points to crit1, the object being assigned to.

The next line, if (this != &c), checks to see whether the address of crit1 is not equal to the address of crit2—that is, it tests if the object isn’t being assigned to itself. Because it’s not, the block associated with the if statement executes.

Inside the if block, delete m\_pName; frees the memory on the heap that crit1’s m\_pName data member pointed to. The line m\_pName = new string(\*(c.m\_pName)); allocates a new chunk of memory on the heap, gets a copy of the string pointed to by the m\_pName data member of crit2, copies the string object to the new heap memory, and points the m\_pName data member of crit1 to this memory. You should follow this logic for all data members that point to memory on the heap.

The last line in the block, m\_Age = c.m\_Age; simply copies the value of the crit2’s m\_Age to crit1’s m\_Age data member. You should follow this simple member- wise copying for all data members that are not pointers to memory on the heap.

Finally, the member function returns a copy of the new crit1 by returning

\*this. You should do the same for any overloaded assignment operator member function you write.

Introducing the Game Lobby Program 315

Back in testAssignmentOp(), I prove that the assignment worked by calling crit1.Greet() and crit2.Greet(). crit1 displays the message I’m crit2 and I’m 9 years old. &m\_pName: 73F2ED48003AF644 while crit2 displays the message I’m crit2 and I’m 9 years old. &m\_pName: 73F2ED48003AF634. The first part of each message, I’m crit2 and I’m 9 years old., is the same and shows that the copying of values worked. The second part of each message is different and shows that each object points to different chunks of memory on the heap, which demon- strates that I avoided shallow copies and have truly independent objects after the assignment.

In the last test of the overloaded assignment operator, I demonstrate what happens when you assign an object to itself. That’s what I do next in the function with the following lines:

Critter crit3("crit", 11); crit3 = crit3;

The preceding assignment statement, crit3 = crit3;, calls the assignment operator member function—operator=()—for crit3. The if statement checks to see whether crit3 is being assigned to itself. Because it is, the member function simply returns a reference to the object through return \*this. You should follow this logic in your own overloaded assignment operator because of potential problems that can arise from only one object being involved in an assignment.

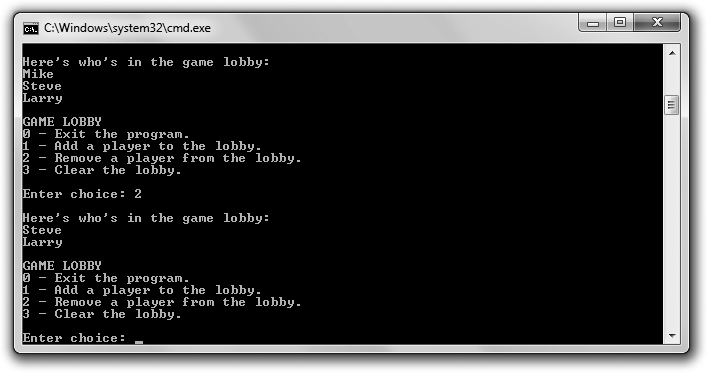
**Hint**

When you have a class with a data member that points to memory on the heap, you should consider overloading the assignment operator for the class.

### [Introducing the Game Lobby Program](#_bookmark11)

The Game Lobby program simulates a game lobby—a waiting area for players, usually in an online game. The program doesn’t actually involve an online component. It creates a single line in which players can wait. The user of the program runs the simulation and has four choices. He can add a person to the lobby, remove a person from the lobby (the first person in line is the first to leave), clear out the lobby, or quit the simulation. Figure 9.11 shows the program in action.

316 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby



**Figure 9.11**

The lobby holds players who are removed in the order in which they were added.

#### [The Player Class](#_bookmark11)

The first thing I do is create a Player class to represent the players who are waiting in the game lobby. Because I don’t know how many players I’ll have in my lobby at one time, it makes sense to use a dynamic data structure. Normally, I’d go to my toolbox of containers from the STL. But I decided to take a different approach in this program and create my own kind of container using dynamically allocated memory that I manage. I didn’t do this because it’s a better programming choice—always see whether you can leverage good work done by other programmers, like the STL—but because it makes for a better game programming example. It’s a great way to really see dynamic memory in action.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 9 folder; the filename is game\_lobby.cpp. Here’s the beginning of the program, which includes the Player class:

//Game Lobby

//Simulates a game lobby where players wait #include <iostream>

Introducing the Game Lobby Program 317

#include <string> using namespace std; class Player

{

public:

Player(const string& name = ""); string GetName() const;

Player\* GetNext() const; void SetNext(Player\* next);

private:

string m\_Name;

Player\* m\_pNext; //Pointer to next player in list

};

Player::Player(const string& name): m\_Name(name),

m\_pNext(0)

{}

string Player::GetName() const

{

return m\_Name;

}

Player\* Player::GetNext() const

{

return m\_pNext;

}

void Player::SetNext(Player\* next)

{

m\_pNext = next;

}

The m\_Name data member holds the name of a player. That’s pretty straightfor- ward, but you might be wondering about the other data member, m\_pNext. It’s a pointer to a Player object, which means that each Player object can hold a name and point to another Player object. You’ll get the point of all this when I talk

318 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

**Figure 9.12**



A Player object can hold a name and point to another Player object.

about the Lobby class. Figure 9.12 provides a visual representation of a Player

object.

The class has a get accessor method for m\_Name and get and set accessor member functions for m\_pNext. Finally, the constructor is pretty simple. It initializes m\_Name to a string object based on what’s passed to the constructor. It also sets m\_pNext to 0, making it a null pointer.

#### [The Lobby Class](#_bookmark11)

The Lobby class represents the lobby or line in which players wait. Here’s the class definition:

class Lobby

{

friend ostream& operator«(ostream& os, const Lobby& aLobby);

public:

Lobby();

~Lobby();

void AddPlayer(); void RemovePlayer(); void Clear();

private:

Player\* m\_pHead;

};

The data member m\_pHead is a pointer that points to a Player object, which represents the first person in line. m\_pHead represents the head of the line.

Because each Player object has an m\_pNext data member, you can link a bunch of Player objects in a *linked list*. Individual elements of linked lists are often

Introducing the Game Lobby Program 319

**Figure 9.13**



Each node holds a name and a pointer to the next player in the list. The first player in line is at the head.

called *nodes*. Figure 9.13 provides a visual representation of a game lobby—a series of player nodes linked with one player at the head of the line.

One way to think about the player nodes is as a group of train cars that carry cargo and are connected. In this case, the train cars carry a name as cargo and are linked through a pointer data member, m\_pNext. The Lobby class allocates memory on the heap for each Player object in the list. The Lobby data member m\_pHead provides access to the first Player object at the head of the list.

The constructor is very simple. It simply initializes the data member m\_pHead to

0, making it a null pointer.

Lobby::Lobby(): m\_pHead(0)

{}

The destructor simply calls Clear(), which removes all the Player objects from the list, freeing the allocated memory.

Lobby::~Lobby()

{

Clear();

}

AddPlayer() instantiates a Player object on the heap and adds it to the end of the list. RemovePlayer() removes the first Player object in the list, freeing the allocated memory.

I declare the function operator<<() a friend of Lobby so that I can send a Lobby

object to cout using the << operator.

320 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

**Trap**

The Lobby class has a data member, m\_pHead, which points to Player objects on the heap. Because of this, I included a destructor that frees all of the memory occupied by the Player objects on the heap instantiated by a Lobby object to avoid any memory leaks when a Lobby object is destroyed. However, I didn’t define a copy constructor or overload the assignment operator in the class. For the Game Lobby program, this isn’t necessary. But if I wanted a more robust Lobby class, I would have defined these member functions.

#### [The Lobby::AddPlayer() Member Function](#_bookmark11)

The Lobby::AddPlayer() member function adds a player to the end of the line in the lobby.

void Lobby::AddPlayer()

{

//create a new player node

cout << "Please enter the name of the new player: "; string name;

cin >> name;

Player\* pNewPlayer = new Player(name);

//if list is empty, make head of list this new player if (m\_pHead == 0)

{

m\_pHead = pNewPlayer;

}

//otherwise find the end of the list and add the player there else

{

Player\* pIter = m\_pHead; while (pIter->GetNext() != 0)

{

pIter = pIter->GetNext();

}

pIter->SetNext(pNewPlayer);

}

}

The first thing the function does is gets the new player’s name from the user and use it to instantiate a new Player object on the heap. Then it sets the object’s pointer data member to the null pointer.

Introducing the Game Lobby Program 321

Next, the function checks to see whether the lobby is empty. If the Lobby object’s data member m\_pHead is 0, then there’s no one in line. If so, the new Player object becomes the head of the line and m\_pHead is set to point to a new Player object on the heap.

If the lobby isn’t empty, the player is added to the end of the line. The function accomplishes this by moving through the list one node at a time, using pIter’s GetNext() member function, until it reaches a Player object whose GetNext() returns 0, meaning that it’s the last node in the list. Then, the function makes that node point to the new Player object on the heap, which has the effect of adding the new object to the end of the list. Figure 9.14 illustrates this process.



**Figure 9.14**



The list of players just before and just after a new player node is added.

322 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

**Trap**

Lobby::AddPlayer() marches through the entire list of Player objects every time it’s called. For small lists this isn’t a problem, but with large lists this inefficient process can become unwieldy. There are more efficient ways to do what this function does. In one of the chapter exercises, your job will be to implement one of these more efficient methods.

#### [The Lobby::RemovePlayer() Member Function](#_bookmark11)

The Lobby:: RemovePlayer() member function removes the player at the head of the line.

void Lobby::RemovePlayer()

{

if (m\_pHead == 0)

{

cout << "The game lobby is empty. No one to remove!\n";

}

else

{

Player\* pTemp = m\_pHead; m\_pHead = m\_pHead->GetNext(); delete pTemp;

}

}

The function tests m\_pHead. If it’s 0, then the lobby is empty and the function displays a message that says so. Otherwise, the first player object in the list is removed. The function accomplishes this by creating a pointer, pTemp, and pointing it to the first Player object in the list. Then the function sets m\_pHead to the next thing in the list—either the next Player object or 0. Finally, the function destroys the Player object pointed to by pTemp. Check out Figure 9.15 for a visual representation of how this works.

#### [The Lobby::Clear() Member Function](#_bookmark11)

The Lobby::Clear() member function removes all of the players from the lobby.

void Lobby::Clear()

{

while (m\_pHead != 0)

{

Introducing the Game Lobby Program 323



**Figure 9.15**



The list of players just before and just after a player node is removed.

RemovePlayer();

}

}

If the list is empty, the loop isn’t entered and the function ends. Otherwise, the loop is entered and the function keeps removing the first Player object in the list by calling RemovePlayer() until there are no more Player objects.

#### [The operator<<() Member Function](#_bookmark11)

The operator<<() member function overloads the << operator so I can display a

Lobby object by sending it to cout.

ostream& operator<<(ostream& os, const Lobby& aLobby)

{

Player\* pIter = aLobby.m\_pHead;

324 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

os << "\nHere’s who’s in the game lobby:\n"; if (pIter == 0)

{

os << "The lobby is empty.\n";

}

else

{

while (pIter != 0)

{

os << pIter->GetName() << endl; pIter = pIter->GetNext();

}

}

return os;

}

If the lobby is empty, the appropriate message is sent to the output stream. Otherwise, the function cycles through all of the players in the list, sending their names to the output stream, using pIter to move through the list.

#### [The main() Function](#_bookmark11)

The main() function displays the players in the lobby, presents the user with a menu of choices, and performs the requested action.

int main()

{

Lobby myLobby; int choice;

do

{

cout << myLobby;

cout << "\nGAME LOBBY\n";

cout << "0 - Exit the program.\n";

cout << "1 - Add a player to the lobby.\n";

cout << "2 - Remove a player from the lobby.\n"; cout << "3 - Clear the lobby.\n";

cout << endl << "Enter choice: "; cin >> choice;

Summary 325

switch (choice)

{

case 0: cout << "Good-bye.\n"; break; case 1: myLobby.AddPlayer(); break; case 2: myLobby.RemovePlayer(); break; case 3: myLobby.Clear(); break;

default: cout << "That was not a valid choice.\n";

}

}

while (choice != 0);

return 0;

}

The function first instantiates a new Lobby object, and then it enters a loop that presents a menu and gets the user’s choice. Then it calls the corresponding Lobby object’s member function. If the user enters an invalid choice, he or she is told so. The loop continues until the user enters 0.

### [Summary](#_bookmark11)

In this chapter, you should have learned the following concepts:

n Aggregation is the combining of objects so that one is part of another.

n Friend functions have complete access to any member of a class.

n Operator overloading allows you to define new meanings for built-in operators as they relate to objects of your own classes.

n The stack is an area of memory that is automatically managed for you and is used for local variables.

n The heap (or free store) is an area of memory that you, the programmer, can use to allocate and free memory.

n The new operator allocates memory on the heap and returns its address.

n The delete operator frees memory on the heap that was previously allocated.

n A dangling pointer points to an invalid memory location. Dereferencing or deleting a dangling pointer can cause your program to crash.

326 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

n A memory leak is an error in which memory that has been allocated becomes inaccessible and can no longer be freed. Given a large enough leak, a program might run out of memory and crash.

n A destructor is a member function that’s called just before an object is destroyed. If you don’t write a destructor of your own, the complier will supply a default destructor for you.

n The copy constructor is a member function that’s invoked when an automatic copy of an object is made. A default copy constructor is supplied for a class if you don’t write one of your own.

n The default copy constructor simply copies the value of each data member to data members with the same names in the copy, producing a member-wise copy.

n Member-wise copying can produce a shallow copy of an object, in which the pointer data members of the copy point to the same chunks of memory as the pointers in the original object.

n A deep copy is a copy of an object that has no chunks of memory in common with the original.

n A default assignment operator member function, which provides only member-wise duplication, is supplied for you if you don’t write one of your own.

n The this pointer is a pointer that all non-static member functions automatically have; it points to the object that was used to call the function.

### [Questions and Answers](#_bookmark11)

Q: Why should you use aggregation?

A: To create more complex objects from other objects. Q: What is composition?

A: A form of aggregation in which the composite object is responsible for the creation and destruction of its object parts. Composition is often called a “uses a” relationship.

Questions and Answers 327

Q: When should I use a friend function?

A: When you need a function to have access to the non-public members of a class.

Q: What is a friend member function?

A: A member function of one class that can access all of the members of another class.

Q: What is a friend class?

A: A class that can access all of the members of another class. Q: Can’t operator overloading become confusing?

A: Yes. Giving too many meanings or unintuitive meanings to operators can lead to code that’s difficult to understand.

Q: What happens when I instantiate a new object on the heap?

A: All of the data members will occupy memory on the heap and not on the stack.

Q: Can I access an object through a constant pointer?

A: Sure. But you can only access constant member functions through a constant pointer.

Q: What’s wrong with shallow copies?

A: Because shallow copies share references to the same chunks of memory, a change to one object will be reflected in another object.

Q: What is a linked list?

A: A dynamic data structure that consists of a sequence of linked nodes. Q: How is a linked list different from a vector?

A: Linked lists permit insertion and removal of nodes at any point in the list but do not allow random access, like vectors. However, the insertion and deletion of nodes in the middle of the list can be more efficient than the insertion and deletion of elements in the middle of vectors.

328 Chapter 9 n Advanced Classes and Dynamic Memory: Game Lobby

Q: Is there a container class from the STL that serves as a linked list? A: Yes, the list class.

Q: Is the data structure used in the Game Lobby program a linked list? A: It shares similarities to a linked list, but it is really a queue.

Q: What’s a queue?

A: A data structure in which elements are removed in the same order in which they were entered. This process is often called first in, first out (FIFO).

Q: Is there a kind of container from the STL that serves as a queue? A: Yes, the queue container adaptor.

### [Discussion Questions](#_bookmark11)

1. What types of game entities could you create with aggregation?
2. Do friend functions undermine encapsulation in OOP?
3. What advantages does dynamic memory offer to game programs?
4. Why are memory leaks difficult errors to track down?
5. Should objects that allocate memory on the heap always be required to free it?

### [Exercises](#_bookmark11)

1. Improve the Lobby class from the Game Lobby program by writing a friend function of the Player class that allows a Player object to be sent to cout. Next, update the function that allows a Lobby object to be sent to cout so that it uses your new function for sending a Player object to cout.
2. The Lobby::AddPlayer() member function from the Game Lobby pro- gram is inefficient because it iterates through all of the player nodes to add a new player to the end of the line. Add an m\_pTail pointer data member to the Lobby class that always points to the last player node in the line and use it to more efficiently add a player.

Exercises 329

1. What’s wrong with the following code?

#include <iostream> using namespace std;

int main()

{

int\* pScore = new int;

\*pScore = 500;

pScore = new int(1000); delete pScore;

pScore = 0;

return 0;

}

*This page intentionally left blank*

|  |  |
| --- | --- |
|  |  |
| [chapter 10](#_bookmark11) | |

# [Inheritance and Polymorphism: Blackjack](#_bookmark11)

Classes give you the perfect way to represent game entities that have attributes and behaviors. But game entities are often related. In this chapter, you’ll learn about inheritance and polymorphism, which give you ways to express those connections and can make defining and using classes even simpler and more intuitive. Specifically, you’ll learn to:

n Derive one class from another

n Use inherited data members and member functions

n Override base class member functions

n Define virtual functions to enable polymorphism

n Declare pure virtual functions to define abstract classes

### [Introducing Inheritance](#_bookmark11)

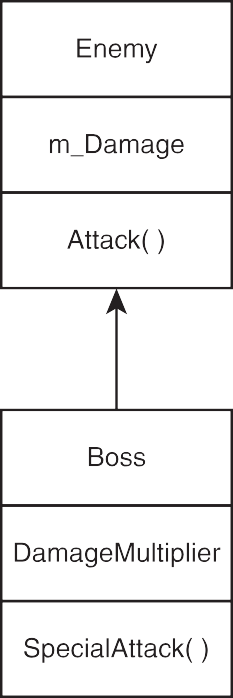
One of the key elements of OOP is *inheritance*, which allows you to *derive* a new class from an existing one. When you do so, the new class automatically *inherits* (or gets) the data members and member functions of an existing class. It’s like getting the work that went into the existing class for free!

Inheritance is especially useful when you want to create a more specialized version of an existing class because you can add data members and member functions to the new class to extend it. For example, imagine you have a class Enemy that defines an enemy in a game with a member function Attack() and a

331

332 Chapter 10 n Inheritance and Polymorphism: Blackjack

data member m\_Damage. You can derive a new class Boss from Enemy for a boss. This means that Boss could automatically have Attack() and m\_Damage without you having to write any code for them at all. Then, to make a boss tough, you could add a member function SpecialAttack() and a data member DamageMultiplier to the Boss class. Take a look at Figure 10.1, which shows the relationship between the Enemy and Boss classes.



**Figure 10.1**

Boss inherits Attack() and m\_Damage from Enemy while defining SpecialAttack() and

m\_DamageMultiplier.

One of the many advantages of inheritance is that you can reuse classes you’ve already written. This reusability produces benefits that include:

n **Less work**. There’s no need to redefine functionality you already have.

Once you have a class that provides the base functionality for other classes, you don’t have to write that code again.

n **Fewer errors**. Once you’ve got a bug-free class, you can reuse it without errors cropping up in it.

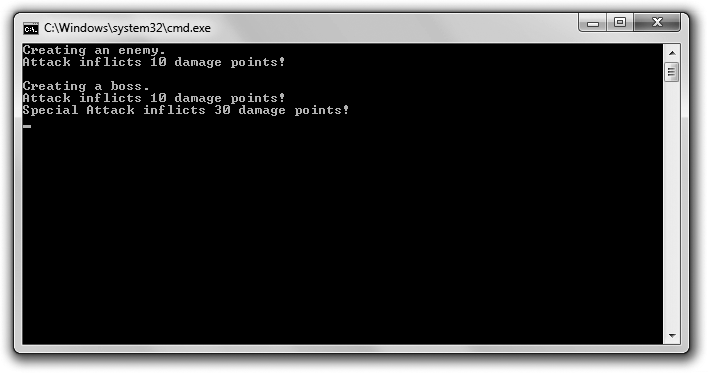
Introducing Inheritance 333

n **Cleaner code**. Because the functionality of base classes exist only once in a program, you don’t have to wade through the same code repeatedly, which makes programs easier to understand and modify.

Most related game entities cry out for inheritance. Whether it’s the series of enemies that a player faces, squadrons of military vehicles that a player commands, or an inventory of weapons that a player wields, you can use inheritance to define these groups of game entities in terms of each other, which results in faster and easier programming.

#### [Introducing the Simple Boss Program](#_bookmark11)

The Simple Boss program demonstrates inheritance. In it,I define a class for lowly enemies, Enemy. From this class, I derive a new class for tough bosses that the player has to face, Boss. Then, I instantiate an Enemy object and call its Attack() member function. Next, I instantiate a Boss object. I’m able to call Attack() for the Boss object because it inherits the member function from Enemy. Finally, I call the Boss object’s SpecialAttack() member function, which I defined in Boss, for a special attack. Since I define SpecialAttack() in Boss, only Boss objects have access to it. Enemy objects don’t have this special attack at their disposal. Figure 10.2 shows the results of the program.



**Figure 10.2**

The Boss class inherits the Attack() member function and then defines its own SpecialAttack()

member function.

334 Chapter 10 n Inheritance and Polymorphism: Blackjack

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 10 folder; the filename is simple\_boss.cpp.

//Simple Boss

//Demonstrates inheritance

#include <iostream> using namespace std;

class Enemy

{

public:

int m\_Damage;

Enemy();

void Attack() const;

};

Enemy::Enemy(): m\_Damage(10)

{}

void Enemy::Attack() const

{

cout << "Attack inflicts " << m\_Damage << " damage points!\n";

}

class Boss : public Enemy

{

public:

int m\_DamageMultiplier;

Boss();

void SpecialAttack() const;

};

Boss::Boss():

m\_DamageMultiplier(3)

{}

Introducing Inheritance 335

void Boss::SpecialAttack() const

{

cout << "Special Attack inflicts " << (m\_DamageMultiplier \* m\_Damage); cout << " damage points!\n";

}

int main()

{

cout << "Creating an enemy.\n"; Enemy enemy1;

enemy1.Attack();

cout << "\nCreating a boss.\n"; Boss boss1;

boss1.Attack(); boss1.SpecialAttack();

return 0;

}

#### [Deriving from a Base Class](#_bookmark11)

I derive the Boss class from Enemy when I define Boss with the following line:

class Boss : public Enemy

Boss is based on Enemy. In fact, Enemy is called the *base class* (or *superclass*) and Boss the *derived class* (or *subclass*). This means that Boss inherits Enemy’s data members and member functions, subject to access controls. In this case, Boss inherits and can directly access m\_Damage and Attack(). It’s as if I defined both m\_Damage and Attack() in Boss.

**Hint**

You might have noticed that I made all of the members of the classes pubic, including their data members. I did this because it makes for the simplest first example of a base and derived class. You also might have noticed that I used the keyword public when deriving Boss from Enemy. For now, don’t worry about this. I’ll cover it all in the next example program, Simple Boss 2.0.

336 Chapter 10 n Inheritance and Polymorphism: Blackjack

To derive classes of your own, follow my example. After the class name in a class definition, put a colon followed by an access modifier (such as public), followed by the name of the base class. It’s perfectly acceptable to derive a new class from a derived class, and sometimes it makes perfect sense to do so. However, to keep things simple, I’m only going to deal with one level of inheritance in this example.

There are actually a few base class member functions that are not inherited by derived classes. They are as follows:

n Constructors

n Copy constructors

n Destructors

n Overloaded assignment operators

You have to write your own versions of these in the derived class.

#### [Instantiating Objects from a Derived Class](#_bookmark11)

In main(), I instantiate an Enemy object and then call its Attack() member function. This works just as you’d expect. The interesting part of the program begins next, when I instantiate a Boss object.

Boss boss1;

After this line of code, I have a Boss object with an m\_Damage data member equal to 10 and an m\_DamageMultiplier data member equal to 3. How did this happen? Although constructors and destructors are not inherited from a base class, they are called when an instance is created or destroyed. In fact, a base class constructor is called before the derived class constructor to create its part of the final object.

In this case, when a Boss object is instantiated, the default Enemy constructor is automatically called and the object gets an m\_Damage data member with a value of 10 (just like any Enemy object would). Then, the Boss constructor is called and finishes off the object by giving it an m\_DamageMultiplier data member with a value of 3. The reverse happens when a Boss object is destroyed at the end of the program. First, the Boss class destructor is called for the object, and then the

Controlling Access under Inheritance 337

Enemy class destructor is called. Because I didn’t define destructors in this program, nothing special happens before the Boss object ceases to exist.

**Hint**

The fact that base class destructors are called for objects of derived classes ensures that each class gets its chance to clean up any part of the object that needs to be taken care of, such as memory on the heap.

#### [Using Inherited Members](#_bookmark11)

Next, I call an inherited member function of the Boss object, which displays the exact same message as enemy1.Attack().

boss1.Attack();

That makes perfect sense because the same code is being executed and both objects have an m\_Damage data member equal to 10. Notice that the function call looks the same as it did for enemy1. The fact that Boss inherited the member function from Enemy makes no difference in how the function is called.

Next, I get Boss to pull out its special attack, which displays the message Special Attack inflicts 30 damage points!

boss1.SpecialAttack();

The thing to notice about this is that SpecialAttack(), declared as a part of Boss, uses the data member m\_Damage, declared in Enemy. That’s perfectly fine. Boss inherits m\_Damage from Enemy and, in this example, the data member works like any other data member in the Boss class.

### [Controlling Access under Inheritance](#_bookmark11)

When you derive one class from another, you can control how much access the derived class has to the base class’ members. For the same reasons that you want to provide only as much access as is necessary to a class’ members to the rest of your program, you want to provide only as much access as is necessary to a class’ members to a derived class. Not coincidentally, you use the same access modifiers that you’ve seen before—public, protected, and private. (Okay, you haven’t seen protected before, but I’ll explain that modifier in the “Using Access Modifiers with Class Members” section.)

338 Chapter 10 n Inheritance and Polymorphism: Blackjack

#### [Introducing the Simple Boss 2.0 Program](#_bookmark11)

The Simple Boss 2.0 program is another version of the Simple Boss program from earlier in this chapter. The new version, Simple Boss 2.0, looks exactly the same to the user, but the code is a little different because I put some restrictions on base class members. If you want to see what the program does, take a look back at Figure 10.2.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 10 folder; the filename is simple\_boss2.cpp.

//Simple Boss 2.0

//Demonstrates access control under inheritance

#include <iostream> using namespace std;

class Enemy

{

public:

Enemy();

void Attack() const;

protected:

int m\_Damage;

};

Enemy::Enemy(): m\_Damage(10)

{}

void Enemy::Attack() const

{

cout << "Attack inflicts " << m\_Damage << " damage points!\n";

}

class Boss : public Enemy

{

public:

Boss();

void SpecialAttack() const;

Controlling Access under Inheritance 339

private:

int m\_DamageMultiplier;

};

Boss::Boss():

m\_DamageMultiplier(3)

{}

void Boss::SpecialAttack() const

{

cout << "Special Attack inflicts " << (m\_DamageMultiplier \* m\_Damage); cout << " damage points!\n";

}

int main()

{

cout << "Creating an enemy.\n"; Enemy enemy1;

enemy1.Attack();

cout << "\nCreating a boss.\n"; Boss boss1;

boss1.Attack(); boss1.SpecialAttack();

return 0;

}

#### [Using Access Modifiers with Class Members](#_bookmark11)

You’ve seen the access modifiers public and private used with class members before, but there’s a third modifier you can use with members of a class— protected. That’s what I use with the data member of Enemy.

protected:

int m\_Damage;

Members that are specified as protected are not accessible outside of the class, except in some cases of inheritance. As a refresher, here are the three levels of member access:

n public members are accessible to all code in a program.

340 Chapter 10 n Inheritance and Polymorphism: Blackjack

n protected members are accessible only in their own class and certain derived classes, depending upon the access level used in inheritance.

n private members are only accessible in their own class, which means they are not directly accessible in any derived class.

#### [Using Access Modifiers When Deriving Classes](#_bookmark11)

When you derive a class from an existing one, you can use an access modifier, such as public, which I used in deriving Boss.

class Boss : public Enemy

Using public derivation means that public members in the base class become public members in the derived class, protected members in the base class become protected members in the derived class, and private members in the base class are inaccessible in the derived class.

**Trick**

Even if base data members are private, you can still use them indirectly through base class member functions. You can even get and set their values if the base class has accessor member functions.

Because Boss inherits from Enemy using the keyword public, Boss inherits Enemy’s public member functions as public member functions. It also means that Boss inherits m\_Damage as a protected data member. The class essentially acts as if I simply copied and pasted the code for these two Enemy class members right into the Boss definition. But through the beauty of inheritance, I didn’t have to do this. The upshot is that the Boss class can access Attack() and m\_Damage().

**Hi n t**

You can derive a new class with the protected and private keywords, but they’re rarely used and are beyond the scope of this book.

### Calling and Overriding Base Class [Member Functions](#_bookmark11)

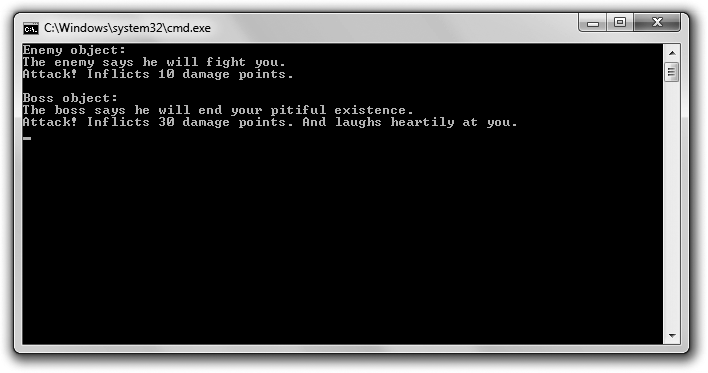
You’re not stuck with every base class member function you inherit in a derived class as is. You have options that allow you to customize how those inherited member functions work in your derived class. You can override them by giving

Calling and Overriding Base Class Member Functions 341

them new definitions in your derived class. You can also explicitly call a base class member function from any member function of your derived class.

#### [Introducing the Overriding Boss Program](#_bookmark11)

The Overriding Boss program demonstrates calling and overriding base class member functions in a derived class. The program creates an enemy that taunts the player and then attacks him. Next, the program creates a boss from a derived class. The boss also taunts the player and attacks him, but the interesting thing is that the inherited behaviors of taunting and attacking are changed for the boss (who is a bit cockier than the enemy). These changes are accomplished through function overriding and calling a base class member function. Figure 10.3 shows the results of the program.



**Figure 10.3**

The Boss class inherits and overrides the base class member functions Taunt() and Attack(), creating new behaviors for the functions in Boss.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 10 folder; the filename is overriding\_boss.cpp.

//Overriding Boss

//Demonstrates calling and overriding base member functions #include <iostream>

using namespace std;

342 Chapter 10 n Inheritance and Polymorphism: Blackjack

class Enemy

{

public:

Enemy(int damage = 10);

void virtual Taunt() const; //made virtual to be overridden void virtual Attack() const; //made virtual to be overridden

private:

int m\_Damage;

};

Enemy::Enemy(int damage): m\_Damage(damage)

{}

void Enemy::Taunt() const

{

cout << "The enemy says he will fight you.\n";

}

void Enemy::Attack() const

{

cout << "Attack! Inflicts " << m\_Damage << " damage points.";

}

class Boss : public Enemy

{

public:

Boss(int damage = 30);

void virtual Taunt() const; //optional use of keyword virtual void virtual Attack() const; //optional use of keyword virtual

};

Boss::Boss(int damage):

Enemy(damage) //call base class constructor with argument

{}

void Boss::Taunt() const //override base class member function

{

cout << "The boss says he will end your pitiful existence.\n";

}

Calling and Overriding Base Class Member Functions 343

void Boss::Attack() const //override base class member function

{

Enemy::Attack(); //call base class member function cout << " And laughs heartily at you.\n";

}

int main()

{

cout << "Enemy object:\n"; Enemy anEnemy; anEnemy.Taunt(); anEnemy.Attack();

cout << "\n\nBoss object:\n"; Boss aBoss;

aBoss.Taunt(); aBoss.Attack();

return 0;

}

#### [Calling Base Class Constructors](#_bookmark11)

As you’ve seen, the constructor for a base class is automatically called when an object of a derived class is instantiated, but you can also explicitly call a base class constructor from a derived class constructor. The syntax for this is a lot like the syntax for a member initialization list. To call a base class constructor from a derived class constructor, after the derived constructor’s parameter list, type a colon followed by the name of the base class, followed by a set of parentheses containing whatever parameters the base class constructor you’re calling needs. I do this in the Boss constructor, which says to explicitly call the Enemy constructor and pass it damage.

Boss::Boss(int damage):

Enemy(damage) //call base class constructor with argument

{}

This allows me to pass the Enemy constructor the value that gets assigned to

m\_Damage, rather than just accepting its default value.

When I first instantiate aBoss in main(), the Enemy constructor is called and passed the value 30, which gets assigned to m\_Damage. Then, the Boss constructor runs (which doesn’t do much of anything) and the object is completed.

344 Chapter 10 n Inheritance and Polymorphism: Blackjack

**Hi n t**

Being able to call a base class constructor is useful when you want to pass specific values to it.

#### [Declaring Virtual Base Class Member Functions](#_bookmark11)

Any inherited base class member function that you expect to be overridden in a derived class should be declared as virtual, using the keyword virtual. When you declare a member function virtual, you provide a way for overridden versions of the member function to work as expected with pointers and references to objects. Since I know that I’ll override Taunt() in the derived class, Boss, I declare Taunt() virtual in my base class, Enemy.

void virtual Taunt() const; //made virtual to be overridden

**Trap**

Although you can override non-virtual member functions, this can lead to behavior you might not expect. A good rule of thumb is to declare any base class member function to be overridden as virtual.

Outside the Enemy class definition, I define Taunt():

void Enemy::Taunt() const

{

cout << "The enemy says he will fight you.\n";

}

Notice that I didn’t use the keyword virtual in the definition. You don’t use

virtual in the definition of a member function, only in its declaration.

Once a member function has been declared as virtual, it’s virtual in any derived class. This means you don’t have to use the keyword virtual in a declaration when you override a virtual member function, but you should use it anyway because it will remind you that the function is indeed virtual. So, when I override Taunt() in Boss, I explicitly declare it as virtual, even though I don’t have to:

void virtual Taunt() const; //optional use of keyword virtual

#### [Overriding Virtual Base Class Member Functions](#_bookmark11)

The next step in overriding is to give the member function a new definition in the derived class. That’s what I do for the Boss class with:

void Boss::Taunt() const //override base class member function

Calling and Overriding Base Class Member Functions 345

{

cout << "The boss says he will end your pitiful existence.\n";

}

This new definition is executed when I call the member function through any Boss object. It replaces the definition of Taunt() inherited from Enemy for all Boss objects. When I call the member function in main() with the following line, the message The boss says he will end your pitiful existence. is displayed.

aBoss.Taunt();

Overriding member functions is useful when you want to change or extend the behavior of base class member functions in derived classes.

**Trap**

Don’t confuse override with overload. When you override a member function, you provide a new definition of it in a derived class. When you overload a function, you create multiple versions of it with different signatures.

**Trap**

When you override an overloaded base class member function, you hide all of the other overloaded versions of the base class member function—meaning that the only way to access the other versions of the member function is to explicitly call the base class member function. So if you override an overloaded member function, it’s a good idea to override every version of the overloaded function.

#### [Calling Base Class Member Functions](#_bookmark11)

You can directly call a base class member function from any function in a derived class. All you have to do is prefix the class name to the member function name with the scope resolution operator. That’s what I do when I define the overridden version of Attack() for the Boss class.

void Boss::Attack() const //override base class member function

{

Enemy::Attack(); //call base class member function cout << " And laughs heartily at you.\n";

}

The code Enemy::Attack(); explicitly calls the Attack() member function of

Enemy. Because the Attack() definition in Boss overrides the class’ inherited

346 Chapter 10 n Inheritance and Polymorphism: Blackjack

version, it’s as if I’ve extended the definition of what it means for a boss to attack. What I’m essentially saying is that when a boss attacks, the boss does exactly what an enemy does and then laughs. When I call the member function for a Boss object in main() with the following line, Boss’ Attack() member function is called because I’ve overloaded Attack().

aBoss.Attack();

The first thing that Boss’ Attack() member function does is explicitly call Enemy’s Attack() member function, which displays the message Attack! Inflicts 30 damage points. Then, Boss’ Attack() member function displays the message And laughs heartily at you.

**Trick**

You can extend the way a member function of a base class works in a derived class by overriding the base class method and then explicitly calling the base class member function from this new definition in the derived class and adding some functionality.

### Using Overloaded Assignment Operators and [Copy Constructors in Derived Classes](#_bookmark11)

You already know how to write an overloaded assignment operator and a copy constructor for a class. However, writing them for a derived class requires a little bit more work because they aren’t inherited from a base class.

When you overload the assignment operator in a derived class, you usually want to call the assignment operator member function from the base class, which you can explicitly call using the base class name as a prefix. If Boss is derived from Enemy, the overloaded assignment operator member function defined in Boss could start:

Boss& operator=(const Boss& b)

{

Enemy::operator=(b); //handles the data members inherited from Enemy

//now take care of data members defined in Boss

The explicit call to Enemy’s assignment operator member function handles the data members inherited from Enemy. The rest of the member function would take care of the data members defined in Boss.

Introducing Polymorphism 347

For the copy constructor, you also usually want to call the copy constructor from a base class, which you can call just like any base class constructor. If Boss is derived from Enemy, the copy constructor defined in Boss could start:

Boss (const Boss& b): Enemy(b) //handles the data members inherited from Enemy

{

//now take care of data members defined in Boss

By calling Enemy’s copy constructor with Enemy(b), you copy that Enemy’s data members into the new Boss object. In the remainder of Boss’ copy constructor, you can take care of copying the data members declared in Boss into the new object.

### [Introducing Polymorphism](#_bookmark11)

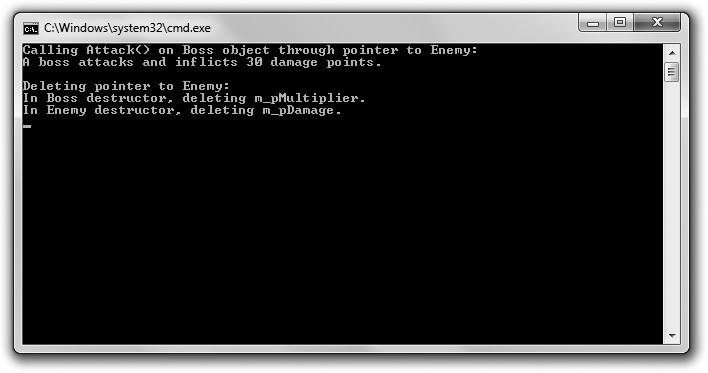
One of the pillars of OOP is *polymorphism*, which means that a member function will produce different results depending on the type of object for which it is being called. For example, suppose you have a group of bad guys that the player is facing, and the group is made of objects of different types that are related through inheritance, such as enemies and bosses. Through the magic of polymorphism, you could call the same member function for each bad guy in the group, say to attack the player, and the type of each object would determine the exact results. The call for the enemy objects could produce one result, such as a weak attack, while the call for bosses could produce a different result, such as a powerful attack. This might sound a lot like overriding, but polymorphism is different because the effect of the function call is dynamic and is determined at run time, depending on the object type. But the best way to understand this isn’t through theoretical discussion; it is through a concrete example.

#### [Introducing the Polymorphic Bad Guy Program](#_bookmark11)

The Polymorphic Bad Guy program demonstrates how to achieve polymorphic behavior. It shows what happens when you use a pointer to a base class to call inherited virtual member functions. It also shows how using virtual destructors ensures that the correct destructors are called for objects pointed to by pointers to a base class. Figure 10.4 shows the results of the program.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 10 folder; the filename is polymorphic\_bad\_guy.cpp.

348 Chapter 10 n Inheritance and Polymorphism: Blackjack



**Figure 10.4**

Through polymorphism the correct member functions and destructors are called for objects pointed to by pointers to a base class.

//Polymorphic Bad Guy

//Demonstrates calling member functions dynamically #include <iostream>

using namespace std;

class Enemy

{

public:

Enemy(int damage = 10); virtual ~Enemy();

void virtual Attack() const;

protected:

int\* m\_pDamage;

};

Enemy::Enemy(int damage)

{

m\_pDamage = new int(damage);

}

Introducing Polymorphism 349

Enemy::~Enemy()

{

cout << "In Enemy destructor, deleting m\_pDamage.\n"; delete m\_pDamage;

m\_pDamage = 0;

}

void Enemy::Attack() const

{

cout << "An enemy attacks and inflicts " << \*m\_pDamage << " damage points.";

}

class Boss : public Enemy

{

public:

Boss(int multiplier = 3); virtual ~Boss();

void virtual Attack() const;

protected:

int\* m\_pMultiplier;

};

Boss::Boss(int multiplier)

{

m\_pMultiplier = new int(multiplier);

}

Boss::~Boss()

{

cout << "In Boss destructor, deleting m\_pMultiplier.\n"; delete m\_pMultiplier;

m\_pMultiplier = 0;

}

void Boss::Attack() const

{

cout << "A boss attacks and inflicts " << (\*m\_pDamage) \* (\*m\_pMultiplier)

<< " damage points.";

}

350 Chapter 10 n Inheritance and Polymorphism: Blackjack

int main()

{

cout << "Calling Attack() on Boss object through pointer to Enemy:\n"; Enemy\* pBadGuy = new Boss();

pBadGuy->Attack();

cout << "\n\nDeleting pointer to Enemy:\n"; delete pBadGuy;

pBadGuy = 0;

return 0;

}

#### [Using Base Class Pointers to Derived Class Objects](#_bookmark11)

An object of a derived class is also a member of the base class. For example, in the Polymorphic Bad Guy program, a Boss object is an Enemy object, too. That makes sense because a boss is really only a specialized kind of enemy. It also makes sense because a Boss object has all of the members of an Enemy object. Okay, so what? Well, because an object of a derived class is also a member of the base class, you can use a pointer to the base class to point to an object of the derived class. That’s what I do in main() with the following line, which instantiates a Boss object on the heap and creates a pointer to Enemy, pBadGuy, that points to the Boss object.

Enemy\* pBadGuy = new Boss();

Why in the world would you want to do this? It’s useful because it allows you to deal with objects without requiring that you know their exact type. For example, you could have a function that accepts a pointer to Enemy that could work with either an Enemy or a Boss object. The function wouldn’t have to know the exact type of object being passed to it; it could work with the object to produce different results depending on the object’s type, as long as derived member functions were declared virtual. Because Attack() is virtual, the correct version of the member function will be called (based on the type of object) and will not be fixed by the type of pointer.

I prove that the behavior will be polymorphic in main(). Remember that pBadGuy is a pointer to Enemy that points to a Boss object. So, the following line calls the Attack() member function of a Boss object through a pointer to Enemy, which correctly results in the Attack() member function defined in Boss being

Introducing Polymorphism 351

called and the text A boss attacks and inflicts 30 damage points. being displayed on the screen.

pBadGuy->Attack();

**Hint**

Virtual functions produce polymorphic behavior through references as well as through pointers.

**Trap**

If you override a non-virtual member function in a derived class and call that member function on a derived class object through a pointer to a base class, you’ll get the results of the base class member function and not the derived class member function definition. This is easier to understand with an example. If in the Polymorphic Bad Guy program I hadn’t declared Attack() as virtual, then when I invoked the member function through a pointer to Enemy on a Boss object with pBadGuy->Attack();, I would have gotten the message An enemy attacks and inflicts 10 damage points. This would have happened as a result of *early binding*, in which the exact member function is bound based on the pointer type—in this case, Enemy. But because Attack() is declared as virtual, the member function call is based on the type of object being pointed to at run time, Boss in this case, not fixed by pointer type. I achieve this polymorphic behavior as the result of *late binding* because Attack() is virtual. The moral of the story is that you should only override virtual member functions.

**Trap**

The benefits of virtual functions aren’t free; there is a performance cost associated with the overhead. Therefore, you should use virtual functions only when you need them.

#### [Defining Virtual Destructors](#_bookmark11)

When you use a pointer to a base class to point to an object of a derived class, you have a potential problem. When you delete the pointer, only the base class’ destructor will be called for the object. This could lead to disastrous results because the derived class’ destructor might need to free memory (as the destructor for Boss does). The solution, as you might have guessed, is to make the base class’ destructor virtual. That way, the derived class’ destructor is called, which (as always) leads to the calling the base class’ destructor, giving every class the chance to clean up after itself.

I put this theory into action when I declare Enemy’s destructor virtual.

virtual ~Enemy();

352 Chapter 10 n Inheritance and Polymorphism: Blackjack

In main(), when I delete the pointer pointing to the Boss object with the following line, the Boss object’s destructor is called, which frees the memory on the heap that m\_pDamageMultiplier points to and displays the message In Boss

destructor, deleting m\_pMultiplier.

delete pBadGuy;

Then, Enemy’s destructor is called, which frees the memory on the heap that m\_pDamage points to and displays the message In Enemy destructor, deleting m\_pDamage. The object is destroyed, and all memory associated with the object is freed.

**Trick**

A good rule of thumb is that if you have any virtual member functions in a class, you should make the destructor virtual, too.

### [Using Abstract Classes](#_bookmark11)

At times you might want to define a class to act as a base for other classes, but it doesn’t make sense to instantiate objects from this class because it’s so generic. For example, suppose you have a game with a bunch of types of creatures running around in it. Although you have a wide variety of creatures, they all have two things in common: They have a health value and they can offer a greeting. So, you could define a class, Creature, as a base from which to derive other classes, such as Pixie, Dragon, Orc, and so on. Although Creature is helpful, it doesn’t really make sense to instantiate a Creature object. It would be great if there were a way to indicate that Creature is a base class only, and not meant for instantiating objects. Well, C++ lets you define a kind of class just like this, called an *abstract class*.

#### [Introducing the Abstract Creature Program](#_bookmark11)

The Abstract Creature program demonstrates abstract classes. In the program, I define an abstract class, Creature, which can be used as a base class for specific creature classes. I define one such class, Orc. Then, I instantiate an Orc object and call a member function to get the orc to grunt hello and another member function to display the orc’s health. Figure 10.5 shows the results of the program.

Using Abstract Classes 353



**Figure 10.5**

The orc is an object instantiated from a class derived from an abstract class for all creatures.

You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 10 folder; the filename is abstract\_creature.cpp.

//Abstract Creature

//Demonstrates abstract classes

#include <iostream> using namespace std;

class Creature //abstract class

{

public:

Creature(int health = 100);

virtual void Greet() const = 0; //pure virtual member function virtual void DisplayHealth() const;

protected:

int m\_Health;

};

Creature::Creature(int health): m\_Health(health)

354 Chapter 10 n Inheritance and Polymorphism: Blackjack

{}

void Creature::DisplayHealth() const

{

cout << "Health: " << m\_Health << endl;

}

class Orc : public Creature

{

public:

Orc(int health = 120); virtual void Greet() const;

};

Orc::Orc(int health):

Creature(health)

{}

void Orc::Greet() const

{

cout << "The orc grunts hello.\n";

}

int main()

{

Creature\* pCreature = new Orc(); pCreature->Greet();

pCreature->DisplayHealth();

return 0;

}

#### [Declaring Pure Virtual Functions](#_bookmark12)

A *pure virtual function* is one to which you don’t need to give a definition. The logic behind this is that there might not be a good definition in the class for the member function. For example, I don’t think it makes sense to define the Greet() function in my Creature class because a greeting really depends on the specific type of creature—a pixie twinkles, a dragon blows a puff of smoke, and an orc grunts.

Using Abstract Classes 355

You specify a pure virtual function by placing an equal sign and a zero at the end of the function header. That’s what I did in Creature with the following line:

virtual void Greet() const = 0; //pure virtual member function

When a class contains at least one pure virtual function, it’s an abstract class. Therefore, Creature is an abstract class. I can use it as the base class for other classes, but I can’t instantiate objects from it.

An abstract class can have data members and virtual functions that are not pure virtual. In Creature, I declare a data member m\_Health and a virtual member function DisplayHealth().

#### [Deriving a Class from an Abstract Class](#_bookmark12)

When you derive a new class from an abstract class, you can override its pure virtual functions. If you override all of its pure virtual functions, then the new class is not abstract and you can instantiate objects from it. When I derive Orc from Creature, I override Creature’s one pure virtual function with the following lines:

void Orc::Greet() const

{

cout << "The orc grunts hello.\n";

}

This means I can instantiate an object from Orc, which is what I do in main()

with the following line:

Creature\* pCreature = new Orc();

The code instantiates a new Orc object on the heap and assigns the memory location of the object to pCreature, a pointer to Creature. Even though I can’t instantiate an object from Creature, it’s perfectly fine to declare a pointer using the class. Like all base class pointers, a pointer to Creature can point to any object of a class derived from Creature, like Orc.

Next, I call Greet(), the pure virtual function that I override in Orc with the following line:

pCreature->Greet();

The correct greeting, The orc grunts hello., is displayed.

356 Chapter 10 n Inheritance and Polymorphism: Blackjack

Finally, I call DisplayHealth(), which I define in Creature.

pCreature->DisplayHealth();

It also displays the proper message, Health: 120.

### [Introducing the Blackjack Game](#_bookmark12)

The final project for this chapter is a simplified version of the casino card game Blackjack (tacky green felt not included). The game works like this: Players are dealt cards with point values. Each player tries to reach a total of 21 without exceeding that amount. Numbered cards count as their face value. An ace counts as either 1 or 11 (whichever is best for the player), and any jack, queen, or king counts as 10.

The computer is the house (the casino) and it competes against one to seven players. At the beginning of the round, all participants (including the house) are dealt two cards. Players can see all of their cards, along with their total. However, one of house’s cards is hidden for the time being.

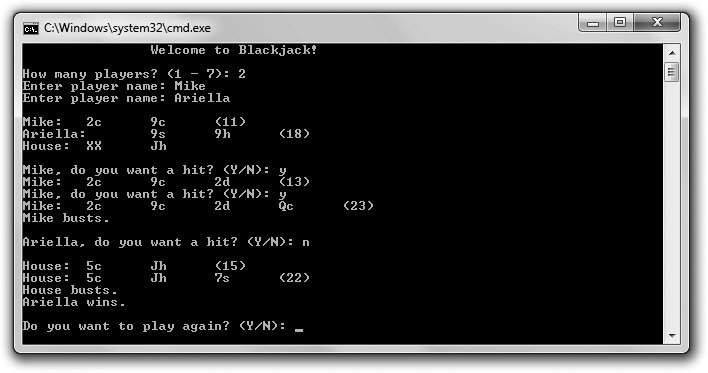
Next, each player gets the chance to take one additional card at a time for as long as he likes. If a player’s total exceeds 21 (known as *busting*), the player loses. After all players have had the chance to take additional cards, the house reveals its hidden card. The house must then take additional cards as long as its total is 16 or less. If the house busts, all players who have not busted win. Otherwise, each remaining player’s total is compared to the house’s total. If the player’s total is greater than the house’s, he wins. If the player’s total is less than the house’s, he loses. If the two totals are the same, the player ties the house (also known as *pushing*). Figure 10.6 shows the game.

#### [Designing the Classes](#_bookmark12)

Before you start coding a project with multiple classes, it is helpful to map them out on paper. You might make a list and include a brief description of each class. Table 10.1 shows my first pass at such a list for the Blackjack game.

To keep things simple, all member functions will be public and all data members will be protected. Also, I’ll use only public inheritance, which means that each derived class will inherit all of its base class members.

Introducing the Blackjack Game 357



**Figure 10.6**

One player wins; the other is not so lucky.

A Blackjack game.

None

GenericPlayer A human Blackjack player.

GenericPlayer The computer player, the house.

Player

House Game

A Blackjack playing card.

A Blackjack hand. A collection of Card objects.

A Blackjack deck. Has extra functionality that Hand doesn’t, such as shuffling and dealing.

A generic Blackjack player. Not a full player, but the common elements of a human player and the computer player.

GenericPlayer Hand

None

None

Hand

Card

Hand Deck

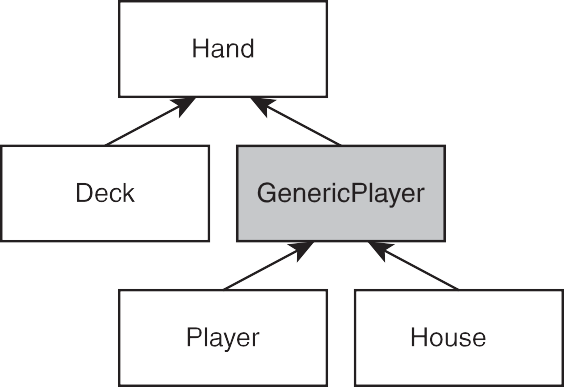
**Table 10.1** Blackjack Classes

**Class Base Class Description**

In addition to describing your classes in words, it helps to draw a family tree of sorts to visualize how your classes are related. That’s what I did in Figure 10.7.

Next, it’s a good idea to get more specific. Ask yourself about the classes. What exactly will they represent? What will they be able to do? How will they work with the other classes?

358 Chapter 10 n Inheritance and Polymorphism: Blackjack



**Figure 10.7**

Inheritance hierarchy of classes for the Blackjack game. GenericPlayer is shaded because it turns out to be an abstract class.

I see Card objects as real-life cards. You don’t copy a card when you deal it from the deck to a hand; you move it. For me, that means Hand will have a data member that is a vector of pointers to Card objects, which will exist on the heap. When a card moves from one Hand to another, it’s really pointers that are being copied and destroyed.

I see players (the human players and the computer) as Blackjack hands with names. That’s why I derive Player and House (indirectly) from Hand. (Another equally valid view is that players have a hand. If I had gone this route, Player and House would have had Hand data members instead of being derived from Hand.)

I define GenericPlayer to house the functionality that Player and House share, as opposed to duplicating this functionality in both classes.

Also, I see the deck as separate from the house. The deck will deal cards to the human players and the computer-controlled house in the same way. This means that Deck will have a member function to deal cards that is polymorphic and will work with either a Player or a House object.

To really flesh things out, you can list the data members and member functions that you think the classes will have, along with a brief description of each. That’s what I do next in Tables 10.2 through 10.8. For each class, I list only the members I define in it. Several classes will, of course, be inherited members from base classes.

Introducing the Blackjack Game 359

**Table 10.2** Card Class

**Member Description**

rank m\_Rank Rank of the card (ace, 2, 3, and so on). rank is an enumeration for all 13 ranks.

suit m\_Suit Suit of the card (clubs, diamonds, hearts, or spades). suit is an enumeration for the four possible suits.

bool m\_IsFaceUp Indicates whether the card is face up. Affects how the card is displayed and the value it has.

int GetValue() Returns the value of the card.

void Flip() Flips a card. Face up becomes face down, and face down becomes face up.

**Table 10.3** Hand Class

**Member Description**

vector<Card\*> m\_Cards Collection of cards. Stores pointers to Card objects.

void Add(Card\* pCard) Adds a card to the hand. Adds a pointer to Card to the vector m\_Cards. void Clear() Clears all cards from the hand. Removes all pointers in the vector m\_Cards,

deleting all associated Card objects on the heap.

int GetTotal() const Returns the total value of the hand.

string m\_Name Generic player’s name.

virtual bool IsHitting() const = 0 Indicates whether the generic player wants another hit. Pure

virtual function.

bool IsBusted() const Indicates whether the generic player is busted.

void Bust() const Announces that the generic player busts.

**Description**

**Member**

**Table 10.4** GenericPlayer Class (Abstract)

virtual bool IsHitting() const Indicates whether the player wants another hit.

void Win() const Announces that the player wins.

void Lose() const Announces that the player loses.

void Push() const Announces that the player pushes.

**Description**

**Table 10.5** Player Class

**Member**

360 Chapter 10 n Inheritance and Polymorphism: Blackjack

**Table 10.6** House Class

**Member**

**Description**

virtual bool IsHitting() const

void FlipFirstCard()

Indicates whether the house is taking another hit.

Flips over the first card.

Creates a standard deck of 52 cards.

Shuffles cards.

Deals one card to a hand.

Gives additional cards to a generic player for as long as the generic player can and wants to hit.

void Populate()

void Shuffle()

void Deal(Hand& aHand)

void AdditionalCards(GenericPlayer& aGenericPlayer)

**Description**

**Table 10.7** Deck Class

**Member**

|  |  |  |
| --- | --- | --- |
|  | **Table 10.8** Game Class |  |
| **Member** | **Description** |
| Deck m\_Deck | A deck of cards. |
| House m\_House | The casino’s hand, the house. |
| vector<Player> m\_Players | Collection of human players. A vector of Player objects. |
| void Play() | Plays a round of Blackjack. |

#### [Planning the Game Logic](#_bookmark12)

The last part of my planning is to map out the basic flow of one round of the game. I wrote some pseudocode for the Game class’ Play() member function. Here’s what I came up with:

*Deal players and the house two initial cards Hide the house’s first card*

*Display players’ and house’s hands Deal additional cards to players*

Introducing the Blackjack Game 361

*Reveal house’s first card*

*Deal additional cards to house If house is busted*

*Everyone who is not busted wins Otherwise*

*For each player*

*If player isn’t busted*

*If player’s total is greater than the house’s total Player wins*

*Otherwise if player’s total is less than house’s total* *Player loses*

*Otherwise Player pushes*

*Remove everyone’s cards*

At this point you know a lot about the Blackjack program and you haven’t even seen a single line of code yet! But that’s a good thing. Planning can be as important as coding (if not more so). Because I’ve spent so much time describing the classes, I won’t describe every part of the code. I’ll just point out significant or new ideas. You can download the code for this program from the Course Technology website [(www.courseptr.com/downloads).](http://www.courseptr.com/downloads) The program is in the Chapter 10 folder; the filename is blackjack.cpp.

**Hint**

The blackjack.cpp file contains seven classes. In C++ programming, it’s common to break up files like this into multiple files, based on individual classes. However, the topic of writing a single program using multiple files is beyond the scope of this book.

#### [The Card Class](#_bookmark12)

After some initial statements, I define the Card class for an individual playing card.

//Blackjack

//Plays a simple version of the casino game of blackjack; for 1 - 7 players

#include <iostream> #include <string> #include <vector> #include <algorithm>

362 Chapter 10 n Inheritance and Polymorphism: Blackjack

#include <ctime>

using namespace std; class Card

{

public:

enum rank {ACE = 1, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING};

enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};

//overloading << operator so can send Card object to standard output friend ostream& operator<<(ostream& os, const Card& aCard);

Card(rank r = ACE, suit s = SPADES, bool ifu = true);

//returns the value of a card, 1 - 11 int GetValue() const;

//flips a card; if face up, becomes face down and vice versa void Flip();

private:

rank m\_Rank; suit m\_Suit; bool m\_IsFaceUp;

};

Card::Card(rank r, suit s, bool ifu): m\_Rank(r), m\_Suit(s), m\_IsFaceUp(ifu)

{}

int Card::GetValue() const

{

//if a cards is face down, its value is 0 int value = 0;

if (m\_IsFaceUp)

{

//value is number showing on card value = m\_Rank;

//value is 10 for face cards if (value > 10)

Introducing the Blackjack Game 363

{

value = 10;

}

}

return value;

}

void Card::Flip()

{

m\_IsFaceUp = !(m\_IsFaceUp);

}

I define two enumerations, rank and suit, to use as the types for the rank and suit data members of the class, m\_Rank and m\_Suit. This has two benefits. First, it makes the code more readable. A suit data member will have a value like CLUBS or HEARTS instead of 0 or 2. Second, it limits the values that these two data members can have. m\_Suit can only store a value from suit, and m\_Rank can only store a value from rank.

Next, I make the overloaded operator<<() function a friend of the class so I can display a card object on the screen.

GetValue() returns a value for a Card object, which can be between 0 and 11. Aces are valued at 11. (I deal with potentially counting them as 1 in the Hand class, based on the other cards in the hand.) A face-down card has a value of 0.

#### [The Hand Class](#_bookmark12)

I define the Hand class for a collection of cards.

class Hand

{

public:

Hand();

virtual ~Hand();

//adds a card to the hand void Add(Card\* pCard);

//clears hand of all cards void Clear();

364 Chapter 10 n Inheritance and Polymorphism: Blackjack

//gets hand total value, intelligently treats aces as 1 or 11 int GetTotal() const;

protected:

vector<Card\*> m\_Cards;

};

Hand::Hand()

{

m\_Cards.reserve(7);

}

Hand::~Hand()

{

Clear();

}

void Hand::Add(Card\* pCard)

{

m\_Cards.push\_back(pCard);

}

void Hand::Clear()

{

//iterate through vector, freeing all memory on the heap vector<Card\*>::iterator iter = m\_Cards.begin();

for (iter = m\_Cards.begin(); iter != m\_Cards.end(); ++iter)

{

delete \*iter;

\*iter = 0;

}

//clear vector of pointers m\_Cards.clear();

}

int Hand::GetTotal() const

{

//if no cards in hand, return 0 if (m\_Cards.empty())

{

return 0;

Introducing the Blackjack Game 365

}

//if a first card has value of 0, then card is face down; return 0 if (m\_Cards[0]->GetValue() = = 0)

{

return 0;

}

//add up card values, treat each ace as 1 int total = 0; vector<Card\*>::const\_iterator iter;

for (iter = m\_Cards.begin(); iter != m\_Cards.end(); ++iter)

{

total += (\*iter)->GetValue();

}

//determine if hand contains an ace bool containsAce = false;

for (iter = m\_Cards.begin(); iter != m\_Cards.end(); ++iter)

{

if ((\*iter)->GetValue() = = Card::ACE)

{

containsAce = true;

}

}

//if hand contains ace and total is low enough, treat ace as 11 if (containsAce && total <= 11)

{

//add only 10 since we’ve already added 1 for the ace total += 10;

}

return total;

}

**Trap**

The destructor of the class is virtual, but notice that I don’t use the keyword virtual outside of the class when I actually define the destructor. You only use the keyword inside the class definition. Don’t worry; the destructor is still virtual.

366 Chapter 10 n Inheritance and Polymorphism: Blackjack

Although I’ve already covered this, I want to point it out again. All of the Card objects will exist on the heap. Any collection of cards, such as a Hand object, will have a vector of pointers to a group of those objects on the heap.

The Clear() member function has an important responsibly. It not only removes all of the pointers from the vector m\_Cards, but it destroys the associated Card objects and frees the memory on the heap that they occupied. This is just like a real-world Blackjack game in which cards are discarded when a round is over. The virtual class destructor calls Clear().

The GetTotal() member function returns the point total of the hand. If a hand contains an ace, it counts it as a 1 or an 11, whichever is best for the player. The program accomplishes this by checking to see whether the hand has at least one ace. If it does, it checks to see whether treating the ace as 11 will put the hand’s point total over 21. If it won’t, then the ace is treated as an 11. Otherwise, it’s treated as a 1.

#### [The GenericPlayer Class](#_bookmark12)

I define the GenericPlayer class for a generic Blackjack player. It doesn’t represent a full player. Instead, it represents the common element of a human player and the computer player.

class GenericPlayer : public Hand

{

friend ostream& operator<<(ostream& os,

const GenericPlayer& aGenericPlayer);

public:

GenericPlayer(const string& name = ""); virtual ~GenericPlayer();

//indicates whether or not generic player wants to keep hitting virtual bool IsHitting() const = 0;

//returns whether generic player has busted - has a total greater than 21 bool IsBusted() const;

//announces that the generic player busts void Bust() const;

Introducing the Blackjack Game 367

protected:

string m\_Name;

};

GenericPlayer::GenericPlayer(const string& name): m\_Name(name)

{}

GenericPlayer::~GenericPlayer()

{}

bool GenericPlayer::IsBusted() const

{

return (GetTotal() > 21);

}

void GenericPlayer::Bust() const

{

cout << m\_Name << " busts.\n";

}

I make the overloaded operator<<() function a friend of the class so I can display GenericPlayer objects on the screen. It accepts a reference to a GenericPlayer object, which means that it can accept a reference to a Player or House object, too.

The constructor accepts a string object for the name of the generic player. The destructor is automatically virtual because it inherits this trait from Hand.

The IsHitting() member function indicates whether a generic player wants another card. Because this member function doesn’t have a real meaning for a generic player, I made it a pure virtual function. Therefore, GenericPlayer becomes an abstract class. This also means that both Player and House need to implement their own versions of this member function.

The IsBusted() member function indicates whether a generic player has busted. Because players and the house bust the same way—by having a total greater than 21—I put the definition in this class.

The Bust() member function announces that the generic player busts. Because busting is announced the same way for players and the house, I put the definition of the member function in this class.

368 Chapter 10 n Inheritance and Polymorphism: Blackjack

#### [The Player Class](#_bookmark12)

The Player class represents a human player. It’s derived from GenericPlayer.

class Player : public GenericPlayer

{

public:

Player(const string& name = ""); virtual ~Player();

//returns whether or not the player wants another hit virtual bool IsHitting() const;

//announces that the player wins void Win() const;

//announces that the player loses void Lose() const;

//announces that the player pushes void Push() const;

};

Player::Player(const string& name): GenericPlayer(name)

{}

Player::~Player()

{}

bool Player::IsHitting() const

{

cout << m\_Name << ", do you want a hit? (Y/N): "; char response;

cin >> response;

return (response = = ’y’ || response = = ’Y’);

}

void Player::Win() const

{

cout << m\_Name << " wins.\n";

}

Introducing the Blackjack Game 369

void Player::Lose() const

{

cout << m\_Name << " loses.\n";

}

void Player::Push() const

{

cout << m\_Name << " pushes.\n";

}

The class implements the IsHitting() member function that it inherits from GenericPlayer. Therefore, Player isn’t abstract. The class implements the member function by asking the human whether he wants to keep hitting. If the human enters y or Y in response to the question, the member function returns true, indicating that the player is still hitting. If the human enters a different character, the member function returns false, indicating that the player is no longer hitting.

The Win(), Lose(), and Push() member functions simply announce that a player has won, lost, or pushed, respectively.

#### [The House Class](#_bookmark12)

The House class represents the house. It’s derived from GenericPlayer.

class House : public GenericPlayer

{

public:

House(const string& name = "House"); virtual ~House();

//indicates whether house is hitting - will always hit on 16 or less virtual bool IsHitting() const;

//flips over first card void FlipFirstCard();

};

370 Chapter 10 n Inheritance and Polymorphism: Blackjack

House::House(const string& name): GenericPlayer(name)

{}

House::~House()

{}

bool House::IsHitting() const

{

return (GetTotal() <= 16);

}

void House::FlipFirstCard()

{

if (!(m\_Cards.empty()))

{

m\_Cards[0]->Flip();

}

else

{

cout << "No card to flip!\n";

}

}

The class implements the IsHitting() member function that it inherits from GenericPlayer. Therefore, House isn’t abstract. The class implements the member function by calling GetTotal(). If the returned total value is less than or equal to 16, the member function returns true, indicating that the house is still hitting. Otherwise, it returns false, indicating that the house is no longer hitting.

FlipFirstCard() flips the house’s first card. This member function is necessary because the house hides its first card at the beginning of the round and then reveals it after all of the players have taken all of their additional cards.

#### [The Deck Class](#_bookmark12)

The Deck class represents a deck of cards. It’s derived from Hand.

class Deck : public Hand

{

Introducing the Blackjack Game 371

public:

Deck();

virtual ~Deck();

//create a standard deck of 52 cards void Populate();

//shuffle cards void Shuffle();

//deal one card to a hand void Deal(Hand& aHand);

//give additional cards to a generic player

void AdditionalCards(GenericPlayer& aGenericPlayer);

};

Deck::Deck()

{

m\_Cards.reserve(52);

Populate();

}

Deck::~Deck()

{}

void Deck::Populate()

{

Clear();

//create standard deck

for (int s = Card::CLUBS; s <= Card::SPADES; ++s)

{

for (int r = Card::ACE; r <= Card::KING; ++r)

{

Add(new Card(static\_cast<Card::rank>(r),

static\_cast<Card::suit>(s)));

}

}

}

void Deck::Shuffle()

372 Chapter 10 n Inheritance and Polymorphism: Blackjack

{

random\_shuffle(m\_Cards.begin(), m\_Cards.end());

}

void Deck::Deal(Hand& aHand)

{

if (!m\_Cards.empty())

{

aHand.Add(m\_Cards.back()); m\_Cards.pop\_back();

}

else

{

cout << "Out of cards. Unable to deal.";

}

}

void Deck::AdditionalCards(GenericPlayer& aGenericPlayer)

{

cout << endl;

//continue to deal a card as long as generic player isn’t busted and

//wants another hit

while ( !(aGenericPlayer.IsBusted()) && aGenericPlayer.IsHitting() )

{

Deal(aGenericPlayer);

cout << aGenericPlayer << endl;

if (aGenericPlayer.IsBusted())

{

aGenericPlayer.Bust();

}

}

}

**Hi n t**

*Type casting* is a way of converting a value of one type to a value of another type. One way to do type casting is to use static\_cast. You use static\_cast to return a value of a new type from a value of another type by specifying the new type you want between < and >, followed by the value from which you want to get a new value between parentheses. Here’s an example that returns the double value 5.0.

static\_cast<double>(5);

Introducing the Blackjack Game 373

Populate() creates a standard deck of 52 cards. The member function loops through all of the possible combinations of Card::suit and Card::rank values. It uses static\_cast to cast the int loop variables to the proper enumerated types defined in Card.

Shuffle() shuffles the cards in the deck. It randomly rearranges the pointers in m\_Cards with random\_shuffle() from the Standard Template Library. This is the reason I include the <algorithm> header file.

Deal() deals one card from the deck to a hand. It adds a copy of the pointer to the back of m\_Cards to the object through the object’s Add() member function. Then, it removes the pointer at the back of m\_Cards, effectively transferring the card. The powerful thing about Deal() is that it accepts a reference to a Hand object, which means it can work equally well with a Player or a House object. And through the magic of polymorphism, Deal() can call the object’s Add() member function without knowing the exact object type.

AdditionalCards() gives additional cards to a generic player until the generic player either stops hitting or busts. The member function accepts reference to a GenericPlayer object so you can pass a Player or House object to it. Again, through the magic of polymorphism, AdditionalCards() doesn’t have to know whether it’s working with a Player or a House object. It can call the IsBusted() and IsHitting() member functions for the object without knowing the object’s type, and the correct code will be executed.

#### [The Game Class](#_bookmark12)

The Game class represents a game of Blackjack.

class Game

{

public:

Game(const vector<string>& names);

~Game();

//plays the game of blackjack void Play();

374 Chapter 10 n Inheritance and Polymorphism: Blackjack

private:

Deck m\_Deck;

House m\_House;

vector<Player> m\_Players;

};

Game::Game(const vector<string>& names)

{

//create a vector of players from a vector of names vector<string>::const\_iterator pName;

for (pName = names.begin(); pName != names.end(); ++pName)

{

m\_Players.push\_back(Player(\*pName));

}

//seed the random number generator srand(static\_cast<unsigned int>(time(0))); m\_Deck.Populate();

m\_Deck.Shuffle();

}

Game::~Game()

{}

void Game::Play()

{

//deal initial 2 cards to everyone vector<Player>::iterator pPlayer; for (int i = 0; i < 2; ++i)

{

for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end();

++pPlayer)

{

m\_Deck.Deal(\*pPlayer);

}

m\_Deck.Deal(m\_House);

}

//hide house’s first card m\_House.FlipFirstCard();

Introducing the Blackjack Game 375

//display everyone’s hand

for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); ++pPlayer)

{

cout << \*pPlayer << endl;

}

cout << m\_House << endl;

//deal additional cards to players

for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); ++pPlayer)

{

m\_Deck.AdditionalCards(\*pPlayer);

}

//reveal house’s first card m\_House.FlipFirstCard(); cout << endl << m\_House;

//deal additional cards to house m\_Deck.AdditionalCards(m\_House);

if (m\_House.IsBusted())

{

//everyone still playing wins

for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end();

++pPlayer)

{

if ( !(pPlayer->IsBusted()) )

{

pPlayer->Win();

}

}

}

else

{

//compare each player still playing to house

for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end();

++pPlayer)

{

if ( !(pPlayer->IsBusted()) )

{

if (pPlayer->GetTotal() > m\_House.GetTotal())

376 Chapter 10 n Inheritance and Polymorphism: Blackjack

{

pPlayer->Win();

}

else if (pPlayer->GetTotal() < m\_House.GetTotal())

{

}

else

{

}

}

}

}

pPlayer->Lose();

pPlayer->Push();

//remove everyone’s cards

for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); ++pPlayer)

{

pPlayer->Clear();

}

m\_House.Clear();

}

The class constructor accepts a reference to a vector of string objects, which represent the names of the human players. The constructor instantiates a Player object with each name. Next, it seeds the random number generator, and then it populates and shuffles the deck.

The Play() member function faithfully implements the pseudocode I wrote earlier about how a round of play should be implemented.

#### [The main() Function](#_bookmark12)

After declaring the overloaded operator«() functions, I write the program’s

main() function.

//function prototypes

ostream& operator<<(ostream& os, const Card& aCard);

ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer);

int main()

{

Introducing the Blackjack Game 377

cout << "\t\tWelcome to Blackjack!\n\n"; int numPlayers = 0;

while (numPlayers < 1 || numPlayers > 7)

{

cout << "How many players? (1 - 7): "; cin >> numPlayers;

}

vector<string> names; string name;

for (int i = 0; i < numPlayers; ++i)

{

cout << "Enter player name: "; cin >> name; names.push\_back(name);

}

cout << endl;

//the game loop Game aGame(names); char again = ’y’;

while (again != ’n’ && again != ’N’)

{

aGame.Play();

cout << "\nDo you want to play again? (Y/N): "; cin >> again;

}

return 0;

}

The main() function gets the names of all the players and puts them into a vector of string objects, and then instantiates a Game object, passing a reference to the vector. The main() function keeps calling the Game object’s Play() member function until the players indicate that they don’t want to play anymore.

#### [Overloading the operator<<() Function](#_bookmark12)

The following function definition overloads the << operator so I can send a Card

object to the standard output.

378 Chapter 10 n Inheritance and Polymorphism: Blackjack

//overloads << operator so Card object can be sent to cout ostream& operator<<(ostream& os, const Card& aCard)

{

const string RANKS[] = {"0", "A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"};

const string SUITS[] = {"c", "d", "h", "s"};

if (aCard.m\_IsFaceUp)

{

os << RANKS[aCard.m\_Rank] << SUITS[aCard.m\_Suit];

}

else

{

os << "XX";

}

return os;

}

The function uses the rank and suit values of the object as array indices. I begin the array RANKS with "0" to compensate for the fact that the value for the rank enumeration defined in Card begins at 1.

The last function definition overloads the << operator so I can send a GenericPlayer

object to the standard output.

//overloads << operator so a GenericPlayer object can be sent to cout ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer)

{

os << aGenericPlayer.m\_Name << ":\t";

vector<Card\*>::const\_iterator pCard; if (!aGenericPlayer.m\_Cards.empty())

{

for (pCard = aGenericPlayer.m\_Cards.begin(); pCard != aGenericPlayer.m\_Cards.end();

++pCard)

{

os << \*(\*pCard) << "\t";

}

if (aGenericPlayer.GetTotal() != 0)

Summary 379

{

}

}

else

{

cout << "(" << aGenericPlayer.GetTotal() << ")";

os << "<empty>";

}

return os;

}

The function displays the generic player’s name and cards, along with the total value of the cards.

### [Summary](#_bookmark12)

In this chapter, you should have learned the following concepts:

n One of the key elements of OOP is inheritance, which allows you to derive a new class from an existing one. The new class automatically inherits data members and member functions from the existing class.

n A derived class does not inherit constructors, copy constructors, destruc- tors, or an overloaded assignment operator.

n Base class constructors are automatically called before the derived class constructor when a derived class object is instantiated.

n Base class destructors are automatically called after the derived class destructor when a derived class object is destroyed.

n Protected members are accessible only in their own class and certain derived classes, depending upon the derivation access level.

n Using public derivation means that public members in the base class become public members in the derived class, protected members in the base class become protected members in the derived class, and private members are (as always) inaccessible.

n You can override base class member functions by giving them new defi- nitions in a derived class.

n You can explicitly call a base class member function from a derived class.

380 Chapter 10 n Inheritance and Polymorphism: Blackjack

n You can explicitly call the base class constructor from a derived class instructor.

n Polymorphism is the quality whereby a member function will produce different results depending on the type of object for which it is called.

n Virtual functions allow for polymorphic behavior.

n Once a member function is defined as virtual, it’s virtual in any derived class.

n A pure virtual function is a function to which you don’t need to give a definition. You specify a pure virtual function by placing an equal sign and a zero at the end of the function header.

n An abstract class has at least one pure virtual member function.

n An abstract class can’t be used to instantiate an object.

### [Questions and Answers](#_bookmark12)

Q: How many levels of inheritance can you have?

A: Theoretically, as many as you want. But as a beginning programmer, you should keep things simple and try not to go beyond a few levels.

Q: Is friendship inherited? That is, if a function is a friend of a base class, is it automatically a friend of a derived class?

A: No.

Q: Can a class have more than one direct base class?

A: Yes. This is called *multiple inheritance*. It’s powerful, but creates its own set of thorny issues.

Q: Why would you want to call a base class constructor from a derived class constructor?

A: So you can control exactly how the base class constructor is called. For example, you might want to pass specific values to the base class constructor.

Q: Are there any dangers in overriding a base class function?

A: Yes. By overriding a base class member function, you hide all of the overloaded version of the function in the base class. However, you can

Questions and Answers 381

still call a hidden base class member function explicitly by using the base class name and the scope resolution operator.

Q: How can I solve this problem of hiding base class functions?

A: One way is to override all of the overloaded version of the base class function.

Q: Why do you usually want to call the assignment operator member function of the base class from the assignment operator member function of a derived class?

A: So that any base class data members can be properly assigned.

Q: Why do you usually want to call the copy constructor of a base class from the copy constructor of a derived class?

A: So that any base class data members can be properly copied.

Q: Why can you lose access to an object’s member functions when you point to it with a base class member?

A: Because non-virtual functions are called based on the pointer type and the object type.

Q: Why not make all member functions virtual, just in case you ever need polymorphic behavior from them?

A: Because there’s a performance cost associated with making member functions virtual.

Q: So when should you make member functions virtual? A: Whenever they may be inherited from a base class.

Q: When should you make a destructor virtual?

A: If you have any virtual member functions in a class, you should make the destructor virtual, too. However, some programmers say that to be safe, you should always make a destructor virtual.

Q: Can constructors be virtual?

A: No. This also means that copy constructors can’t be declared as virtual either. Q: In OOP, what is slicing?

A: Slicing is cutting off part of an object. Assigning an object of a derived class to a variable of a base class is legal, but you slice the object, losing the data

382 Chapter 10 n Inheritance and Polymorphism: Blackjack

members declared in the derived class and losing access to member functions of the derived class.

Q: What good are abstract classes if you can’t instantiate any objects from them?

A: Abstract classes can be very useful. They can contain many common class members that other classes will inherit, which saves you the effort of defining those members over and over again.

### [Discussion Questions](#_bookmark12)

1. What benefits does inheritance bring to game programming?
2. How does polymorphism expand the power of inheritance?
3. What kinds of game entities might it make sense to model through inheritance?
4. What kinds of game-related classes would be best implemented as abstract?
5. Why is it advantageous to be able to point to a derived class object with a base class pointer?

### [Exercises](#_bookmark12)

1. Improve the Simple Boss 2.0 program by adding a new class, FinalBoss, that is derived from the Boss class. The FinalBoss class should define a new method, MegaAttack(), that inflicts 10 times the amount of damage as the SpecialAttack() method does.
2. Improve the Blackjack game program by forcing the deck to repopulate before a round if the number of cards is running low.
3. Improve the Abstract Creature program by adding a new class, OrcBoss, that is derived from Orc. An OrcBoss object should start off with 180 for its health data member. You should also override the virtual Greet() member function so that it displays: The orc boss growls hello.

|  |  |
| --- | --- |
|  |  |
| [appendix A](#_bookmark12) | |

# [Creating Your First C++ Program](#_bookmark12)

Follow these steps to write, save, compile, and run your first program using Microsoft’s Visual Cþþ 2010 Express, a popular and free IDE for the Windows platform.

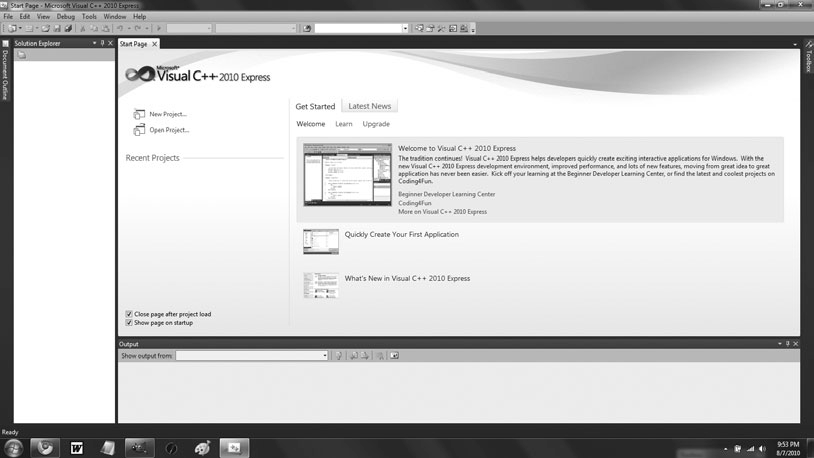
1. Download Visual Cþþ 2010 Express from [http://www.microsoft.com/ express/downloads.](http://www.microsoft.com/express/downloads)
2. Install Visual Cþþ 2010 Express, accepting the default options.
3. Launch Visual Cþþ 2010 Express. You should see what appears in Figure A.1.
4. From the application menu, select File, New, Project. In the New Project dialog that appears, select Win32 from the Installed Templates pane and select Win32 Console Application from the pane to the right. In the Name field, type **game\_over**. In the Location field, browse to the location to save your project by clicking the Browse button. I recommend creating a new folder for the project. (I store my project in C:\Users\Mike\Desk- top\game\_over\.) Last but not least, make sure the check box is checked for Create directory for solution. Your New Project dialog should look similar to the one in Figure A.2.

**Hint**

It’s generally a good idea to store each project in its own folder.

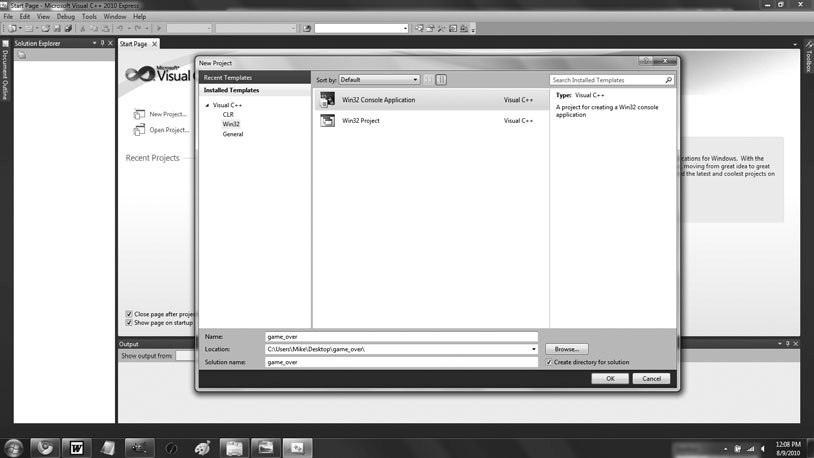
383

384 Appendix A n Creating Your First C++ Program



**Figure A.1**

Visual Cþþ 2010 Express on startup.



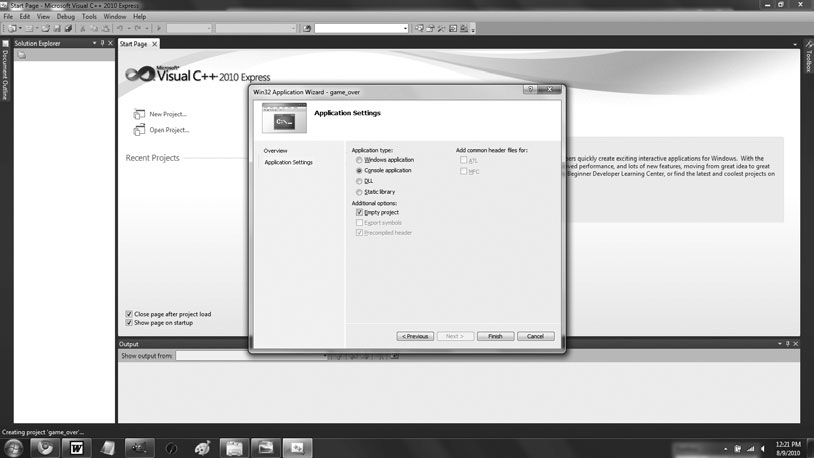
**Figure A.2**

The New Project dialog, filled out.

Creating Your First C++ Program 385

1. With the New Project dialog filled out, click the Okay button. This will bring up the Win32 Application Wizard—Overview. Click the Next button. This will take you to the Win32 Application Wizard—

Application Settings. Under Additional options, check the check box for Empty project. Your screen should look like Figure A.3.



**Figure A.3**

The Win32 Application Wizard – Application Settings, defining an empty project.

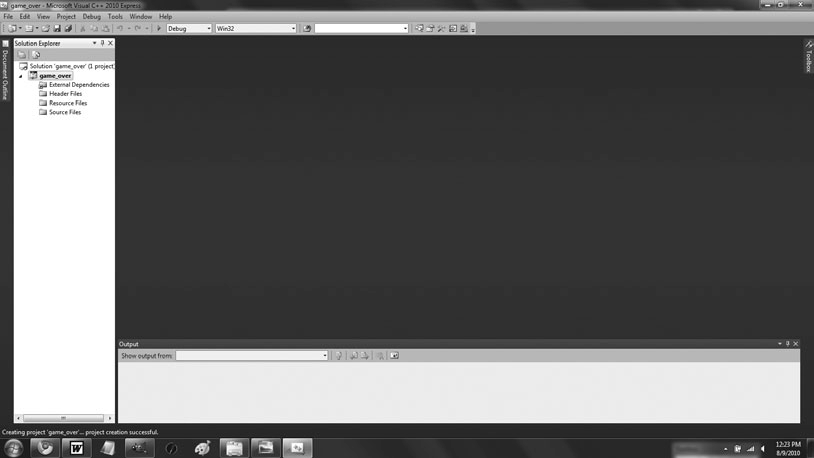
1. In the Win32 Application Wizard—Application Settings, click the Finish button. This will create and open a new solution for your project, as pictured in Figure A.4.

**Hint**

If the Solution Explorer is not displayed, from the application menu, select View, Other Windows, Solution Explorer.

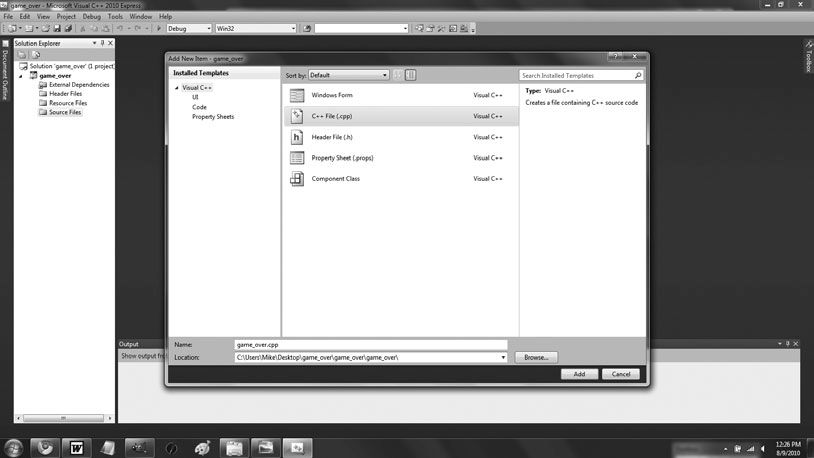
1. In the Solution Explorer, right-click the Source Files folder. From the menu that appears, select Add, New Item. In the Add New Item dialog that appears, select Cþþ File (.cpp). In the Name field, type **game\_over.cpp**. Check out Figure A.5 for a completed Add New Item dialog image.

386 Appendix A n Creating Your First C++ Program



**Figure A.4**

Your newly created project.



**Figure A.5**

The Add New Item dialog, filled out.

Creating Your First C++ Program 387

1. In the Add New Item dialog, click the Add button. The empty Cþþ file named game\_over.cpp appears, ready for editing. In the game\_over.cpp Cþþ file, type the following:

// Game Over

// A first C++ program #include <iostream>

int main()

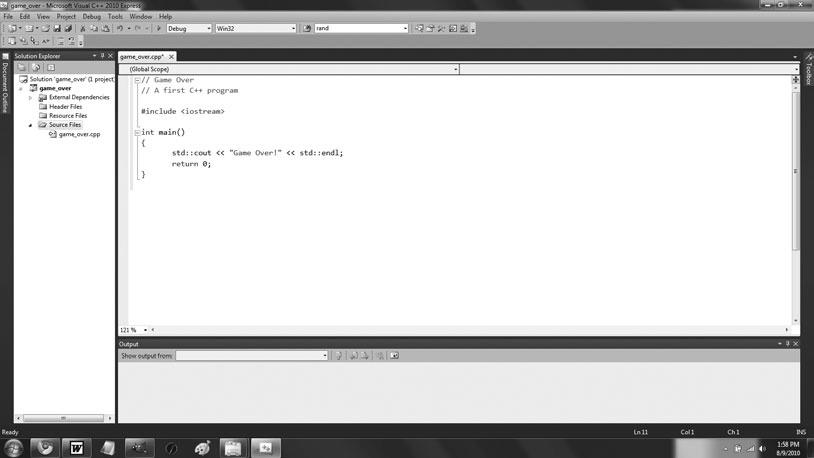
{

std::cout << "Game Over!" << std::endl; return 0;

}

Your screen should look like Figure A.6.

1. From the application menu, select File, Save.

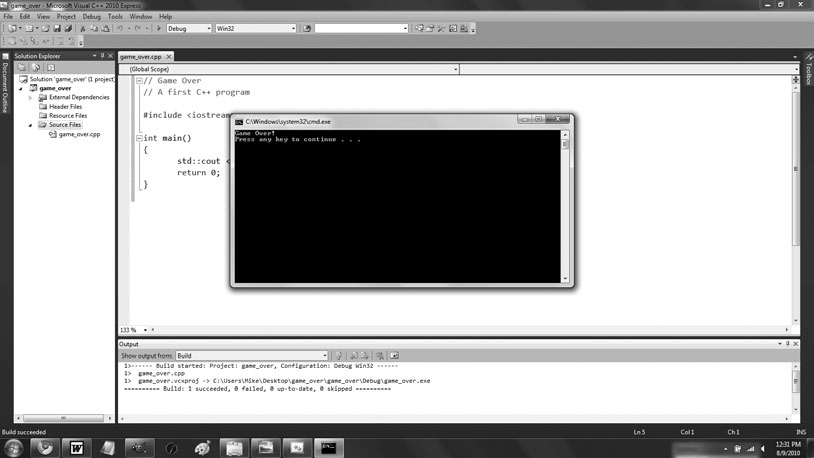


**Figure A.6**

Your new Cþþ file, edited.

388 Appendix A n Creating Your First C++ Program

1. From the application menu, select Debug, Build Solution.
2. Press CtrlþF5 to run the project and enjoy the fruits of your labor. You should see the results shown in Figure A.7.



**Figure A.7**

The big payoff: seeing your program run.

Congratulations! You’ve written, saved, compiled, and run your first Cþþ program.

**Hi n t**

For more detailed information about Microsoft Visual Cþþ 2010 Express, please see its documentation.

|  |  |
| --- | --- |
|  |  |
| [appendix B](#_bookmark12) | |

# [Operator Precedence](#_bookmark12)

|  |  |  |
| --- | --- | --- |
| C++ Operator Precedence | | |
| **Precedence Level** | **Operator** | **Description** |
| 17 | :: | Scope resolution |
| 16 | -> | Indirect member selection |
| 16 | . | Member selection |
| 16 | [ ] | Array index |
| 16 | ( ) | Function call |
| 16 | ( ) | Type construction |
| 16 | sizeof | Size in bytes |
| 15 | ++ | Increment |
| 15 | - - | Decrement |
| 15 | ~ | Bitwise NOT |
| 15 | ! | Logical NOT |
| 15 | + | Unary plus |
| 15 | - | Unary minus |
| 15 | \* | Dereference |
| 15 | & | Address-of |
| 15 | ( ) | Cast |
| 15 | new | Acquire memory on the heap |
| 15 | delete | Release memory on the heap |
| 14 | ->\* | Indirect member pointer selector |
| 14 | .\* | Member pointer selector |
| 13 | \* | Multiplication |

(*Continued*)

389

390 Appendix B n Operator Precedence

|  |  |  |
| --- | --- | --- |
| C++ Operator Precedence (*Continued )* | | |
| **Precedence Level** | **Operator** | **Description** |
| 13 | / | Division |
| 13 | % | Modulus |
| 12 | + | Addition |
| 12 | - | Subtraction |
| 11 | << | Bitwise shift left |
| 11 | >> | Bitwise shift right |
| 10 | < | Less than |
| 10 | <= | Less than or equal to |
| 10 | > | Greater than |
| 10 | >= | Greater than or equal to |
| 9 | == | Equal to |
| 9 | != | Not equal to |
| 8 | & | Bitwise AND |
| 7 | ^ | Bitwise XOR |
| 6 | | | Bitwise OR |
| 5 | && | Logical AND |
| 4 | || | Logical OR |
| 3 | ?: | Conditional operator |
| 2 | = | Assignment |
| 2 | \*= | Multiply and assign |
| 2 | /= | Divide and assign |
| 2 | %= | Modulus and assign |
| 2 | += | Add and assign |
| 2 | -= | Subtract and assign |
| 2 | <<= | Bitwise shift left and assign |
| 2 | >>= | Bitwise shift right and assign |
| 2 | &= | Bitwise AND and assign |
| 2 | |= | Bitwise OR and assign |
| 2 | ^= | Bitwise XOR and assign |
| 1 | , | Comma operator |

|  |  |
| --- | --- |
|  |  |
| [appendix C](#_bookmark12) | |

# [Keywords](#_bookmark12)

This appendix contains a list of Cþþ keywords.

and asm auto bitand bitor bool break case catch char class compl const

const\_cast continue default

delete do double

dynamic\_cast else

enum explicit extern false float for friend goto

if inline int

long mutable namespace new

not not\_eq operator or

or\_eq private protected public register

reinterpret\_cast return

short

391

392 Appendix C n Keywords

signed sizeof static static\_cast struct switch template this

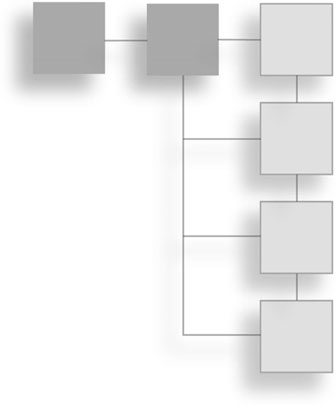
throw true try typedef typeid

typename union unsigned

using virtual void volatile wchar\_t while xor xor\_eq

|  |  |
| --- | --- |
|  |  |
| [appendix D](#_bookmark12) | |

# [ASCII Chart](#_bookmark12)

(*Continued*)

|  |  |  |
| --- | --- | --- |
| ASCII Chart |  | |
| **Decimal** | **Hexadecimal** | **Character** |
| 0 | 00 | NUL |
| 1 | 01 | SOH |
| 2 | 02 | STX |
| 3 | 03 | ETX |
| 4 | 04 | EOT |
| 5 | 05 | ENQ |
| 6 | 06 | ACK |
| 7 | 07 | BEL |
| 8 | 08 | BS |
| 9 | 09 | HT |
| 10 | 0A | LF |
| 11 | 0B | VT |
| 12 | 0C | FF |
| 13 | 0D | CR |
| 14 | 0E | SO |
| 15 | 0F | SI |
| 16 | 10 | DLE |
| 17 | 11 | DC1 |
| 18 | 12 | DC2 |
| 19 | 13 | DC3 |
| 20 | 14 | DC4 |

393

394 Appendix D n ASCII Chart

|  |  |  |
| --- | --- | --- |
| ASCII Chart (*Continued )* | | |
| **Decimal** | **Hexadecimal** | **Character** |
| 21 | 15 | NAK |
| 22 | 16 | SYM |
| 23 | 17 | ETB |
| 24 | 18 | CAN |
| 25 | 19 | EM |
| 26 | 1A | SUB |
| 27 | 1B | ESC |
| 28 | 1C | FS |
| 29 | 1D | GS |
| 30 | 1E | RS |
| 31 | 1F | US |
| 32 | 20 | SP |
| 33 | 21 | ! |
| 34 | 22 | " |
| 35 | 23 | # |
| 36 | 24 | $ |
| 37 | 25 | % |
| 38 | 26 | & |
| 39 | 27 | ’ |
| 40 | 28 | ( |
| 41 | 29 | ) |
| 42 | 2A | \* |
| 43 | 2B | + |
| 44 | 2C | , |
| 45 | 2D | - |
| 46 | 2E | . |
| 47 | 2F | / |
| 48 | 30 | 0 |
| 49 | 31 | 1 |
| 50 | 32 | 2 |
| 51 | 33 | 3 |
| 52 | 34 | 4 |
| 53 | 35 | 5 |
| 54 | 36 | 6 |
| 55 | 37 | 7 |
| 56 | 38 | 8 |
| 57 | 39 | 9 |

(*Continued*)

ASCII Chart 395

|  |  |  |
| --- | --- | --- |
| 58 | 3A | : |
| 59 | 3B | ; |
| 60 | 3C | < |
| 61 | 3D | = |
| 62 | 3E | > |
| 63 | 3F | ? |
| 64 | 40 | @ |
| 65 | 41 | A |
| 66 | 42 | B |
| 67 | 43 | C |
| 68 | 44 | D |
| 69 | 45 | E |
| 70 | 46 | F |
| 71 | 47 | G |
| 72 | 48 | H |
| 73 | 49 | I |
| 74 | 4A | J |
| 75 | 4B | K |
| 76 | 4C | L |
| 77 | 4D | M |
| 78 | 4E | N |
| 79 | 4F | O |
| 80 | 50 | P |
| 81 | 51 | Q |
| 82 | 52 | R |
| 83 | 53 | S |
| 84 | 54 | T |
| 85 | 55 | U |
| 86 | 56 | V |
| 87 | 57 | W |
| 88 | 58 | X |
| 89 | 59 | Y |
| 90 | 5A | Z |
| 91 | 5B | [ |
| 92 | 5C | \ |
| 93 | 5D | ] |
| 94 | 5E | ^ |

(*Continued*)

ASCII Chart *(Continued )*

**Decimal**

**Hexadecimal**

**Character**

396 Appendix D n ASCII Chart

ASCII Chart (*Continued )*

**Decimal**

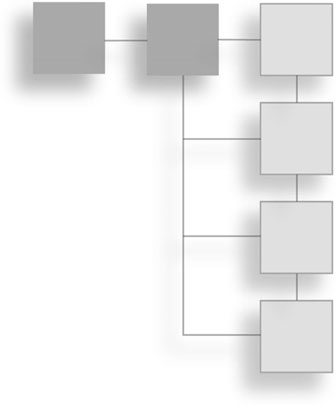
**Hexadecimal**

**Character**

|  |  |  |
| --- | --- | --- |
| 95 | 5F | \_ |
| 96 | 60 | ` |
| 97 | 61 | a |
| 98 | 62 | b |
| 99 | 63 | c |
| 100 | 64 | d |
| 101 | 65 | e |
| 102 | 66 | f |
| 103 | 67 | g |
| 104 | 68 | h |
| 105 | 69 | i |
| 106 | 6A | j |
| 107 | 6B | k |
| 108 | 6C | l |
| 109 | 6D | m |
| 110 | 6E | n |
| 111 | 6F | o |
| 112 | 70 | p |
| 113 | 71 | q |
| 114 | 72 | r |
| 115 | 73 | s |
| 116 | 74 | t |
| 117 | 75 | u |
| 118 | 76 | v |
| 119 | 77 | w |
| 120 | 78 | x |
| 121 | 79 | y |
| 122 | 7A | z |
| 123 | 7B | { |
| 124 | 7C | | |
| 125 | 7D | } |
| 126 | 7E | ~ |
| 127 | 7F | DEL |

|  |  |
| --- | --- |
|  |  |
| [appendix E](#_bookmark12) | |

# [Escape Sequences](#_bookmark12)

397

Single quote

Double quote Backslash Null character System bell Backspace Formfeed Newline

Carriage return Horizontal tab Vertical tab

Hexadecimal number

\’

\"

\\

\0

\a

\b

\f

\n

\r

\t

\v

\x

**Description**

Escape Sequences

**Escape Sequence**

*This page intentionally left blank*

- **(subtraction) operator,** [**14**](#_bookmark54)

[INDEX](#_bookmark12)

**! (NOT) operator, 61*tbl*,** [**66**](#_bookmark176)**–**[**67,**](#_bookmark178) [**77**](#_bookmark201)

**!= (not equal to) operator, 40*tbl***

**" (hash mark) symbol,** [**7,**](#_bookmark30) [**35**](#_bookmark106)

**% (modulus operator),** [**15,**](#_bookmark57) [**72**](#_bookmark190)

**& (reference operator),** [**189,**](#_bookmark454) [**217**](#_bookmark514) **&& (AND) operator, 61*tbl*,** [**77**](#_bookmark201)

* **(dereference operator),** [**128,**](#_bookmark307) [**146,**](#_bookmark354) [**230**](#_bookmark542)
* **(multiplication) operator,** [**14**](#_bookmark54)

**. (member selection operator),** [**88,**](#_bookmark220) [**259**](#_bookmark599)

**/ (forward slash),** [**14**](#_bookmark54)

**:: (scope resolution operator), 9,** [**125,**](#_bookmark302) [**259,**](#_bookmark599) [**345**](#_bookmark753)

**; (semicolon)**

if statement, [43](#_bookmark126)

terminating statements, [9](#_bookmark38)

**[ ] (subscripting operator),** [**93,**](#_bookmark230) [**99,**](#_bookmark243)

[**121,**](#_bookmark292) [**147**](#_bookmark357)

**|| (OR) operator,** [**61*tbl***](#_bookmark168)**,** [**66**](#_bookmark176)

þ **(addition) operator,** [**14,**](#_bookmark54) [**92**](#_bookmark227)

þþ **(increment operator),** [**27**](#_bookmark89)**–**[**28**](#_bookmark90)

*<* **(less than) operator, 40*tbl***

*<<* **(output operator),** [**9**](#_bookmark38)

*<***= (less than or equal to) operator, 40*tbl***

**= (assignment operator),** [**21,**](#_bookmark74) [**44,**](#_bookmark129)

[**313**](#_bookmark697)**–**[**315,**](#_bookmark699) [**346**](#_bookmark755)**–**[**347**](#_bookmark758)

**== (equal to) operator,** [**40*tbl***](#_bookmark119)**,** [**44**](#_bookmark129)

**> (greater than) operator,** [**40*tbl***](#_bookmark119)

**>= (greater than or equal to) operator,** [**40*tbl***](#_bookmark119)

**64-element array,** [**103**](#_bookmark253)

##### A

**abstract classes,** [**352**](#_bookmark764)**–**[**356,**](#_bookmark772) [**380,**](#_bookmark802) [**382**](#_bookmark806)

Abstract Creature program, [352](#_bookmark764)–[354](#_bookmark767)

declaring pure virtual functions, [354](#_bookmark767)–[355](#_bookmark770)

deriving classes from, [355](#_bookmark770)–[356](#_bookmark772) virtual functions and, [355](#_bookmark770)

**Abstract Creature program,** [**352**](#_bookmark764)**–**[**354**](#_bookmark767)

**abstraction**

encapsulation and, [160](#_bookmark383)

functions, [155](#_bookmark374)

**access control,** [**337**](#_bookmark733)**–**[**340**](#_bookmark741)

Simple Boss 2.0 program, [338](#_bookmark736)–[339](#_bookmark737)

using access modiﬁers when deriving classes, [340](#_bookmark741)

using access modiﬁers with class members, [339](#_bookmark737)–[340](#_bookmark741)

**access modiﬁers**

using when deriving classes, [340](#_bookmark741) using with class members,

[339](#_bookmark737)–[340](#_bookmark741)

**access\_global() function,** [**168**](#_bookmark399) **accessing**

array elements, [246](#_bookmark574)

data members, [259](#_bookmark599)–[260](#_bookmark603)

global variables, [168](#_bookmark399) member functions of array

element, [100](#_bookmark245)

member functions of vector element, [129](#_bookmark310)–[130](#_bookmark312)

reference values, [190](#_bookmark456)

static data members, [272](#_bookmark626)–[273](#_bookmark629)

**accessor member functions,** [**267–**](#_bookmark616)[**268,**](#_bookmark618) [**282**](#_bookmark648)

**action statement, for loop,** [**82**](#_bookmark211) **Add() member function,** [**292**](#_bookmark665) **Add New Item dialog box,**

[**386**](#_bookmark812)**ﬁ**[**g**](#_bookmark812)

**addition (**þ**) operator,** [**14,**](#_bookmark54) [**92**](#_bookmark227)

**address, variable,** [**249**](#_bookmark581)

**advanced classes**

aggregation, [287](#_bookmark657)–[292](#_bookmark665) container data members,

[291](#_bookmark663)–[292](#_bookmark665)

Critter Farm program, [288](#_bookmark660)–[290](#_bookmark661)

object data members, [290](#_bookmark661)–[291](#_bookmark663)

friend functions creating, [295](#_bookmark670)

Friend Critter program, [292](#_bookmark665)–[294](#_bookmark668)

Game Lobby program, [315](#_bookmark699)–[325](#_bookmark715)

Lobby class, [318](#_bookmark703)–[320](#_bookmark707)

Lobby::AddPlayer() member function, [320](#_bookmark707)–[322](#_bookmark709)

Lobby::Clear() member function, [322](#_bookmark709)–[323](#_bookmark711)

Lobby::RemovePlayer()

member function, [322](#_bookmark709)

main() function, [324](#_bookmark713)–[325](#_bookmark715)

operator<<() member function, [323](#_bookmark711)–[324](#_bookmark713)

Player class, [316](#_bookmark701)–[318](#_bookmark703)

operator overloading, [295](#_bookmark670)–[296](#_bookmark673) **again character, looping,** [**57**](#_bookmark157)**–**[**58**](#_bookmark159) **aggregation,** [**287**](#_bookmark657)**–**[**292**](#_bookmark665)

container data members, [291](#_bookmark663)–[292](#_bookmark665)

Critter Farm program, [288](#_bookmark660)–[290](#_bookmark661)

deﬁned, [325](#_bookmark716)

object data members, [290](#_bookmark661)–[291](#_bookmark663)

**AI (Artiﬁcial Intelligence),** [**203**](#_bookmark488)

399

400 Index

**algorithms,** [**131**](#_bookmark315)**–**[**135,**](#_bookmark323) [**146**](#_bookmark354)

deﬁned, [116](#_bookmark281)

ﬁnd(), [134](#_bookmark320)

High Scores program, [131](#_bookmark315)–[133](#_bookmark317)

preparing to use, [133](#_bookmark317)

random\_shufﬂe(), [134](#_bookmark320)–[135](#_bookmark323)

sort(), [135](#_bookmark323)

**allocating dynamic memory,** [**296**](#_bookmark673)**–**[**303**](#_bookmark685)

avoiding memory leaks, [301](#_bookmark682)–[303](#_bookmark685)

delete operator, [300](#_bookmark680)–[301](#_bookmark682)

Heap program, [297](#_bookmark677)–[299](#_bookmark679)

new operator, [299](#_bookmark679)–[300](#_bookmark680)

**altering**

global variables, [169](#_bookmark402)

iterators, [128](#_bookmark307)

object through returning pointers, [243](#_bookmark568)–[244](#_bookmark571)

object through returning references, [202](#_bookmark485)

reference values, [190](#_bookmark456)–[191](#_bookmark459) value of variable, [26](#_bookmark85)

**American Standard Code for Information Interchange (ASCII),** [**22,**](#_bookmark76) [**393**](#_bookmark821)**–**[**396**](#_bookmark822)

**AND (&&) operator,** [**61*tbl***](#_bookmark168)**,** [**77**](#_bookmark201)

**announceWinner()function, Tic-Tac-Toe game,** [**217**](#_bookmark514)

**ANSI (ANSI/ISO) standard,** [**5**](#_bookmark25) **argument variable,** [**191,**](#_bookmark459) [**197**](#_bookmark472)

**arguments,** [**70**](#_bookmark185)

versus parameters, [184](#_bookmark441) passing references to alter,

[191](#_bookmark459)–[194](#_bookmark465)

**arithmetic operators,** [**13**](#_bookmark52)**–**[**15**](#_bookmark57)

adding, [14](#_bookmark54)

Expensive Calculator program, [13](#_bookmark52)–[14](#_bookmark54)

ﬂoating point division, [14](#_bookmark54)–[15](#_bookmark57) integer division, [14](#_bookmark54)–[15](#_bookmark57)

modulus operator, [15](#_bookmark57)

multiplying, [14](#_bookmark54)

order of operations, [15](#_bookmark57) subtracting, [14](#_bookmark54)

**Array Passer program,** [**244**](#_bookmark571)**–**[**246**](#_bookmark574) **arrays,** [**96**](#_bookmark238)**–**[**106**](#_bookmark258)

accessing member functions of array element, [100](#_bookmark245)

bounds checking, [100](#_bookmark245)–[101](#_bookmark248)

creating, [98](#_bookmark241)–[99](#_bookmark243)

Hero's Inventory program, [96](#_bookmark238)–[98](#_bookmark241)

indexing, [99](#_bookmark243)–[100](#_bookmark245)

multidimensional, [103](#_bookmark253)–[106](#_bookmark258)

creating, [105](#_bookmark255)

indexing, [105](#_bookmark255)–[106](#_bookmark258) Tic-Tac-Toe Board

program, [103](#_bookmark253)–[105](#_bookmark255)

pointers and, [244](#_bookmark571)–[248](#_bookmark578) Array Passer program,

[244](#_bookmark571)–[246](#_bookmark574)

constant pointer, [246](#_bookmark574)–[247](#_bookmark576) passing and returning,

[247](#_bookmark576)–[248](#_bookmark578)

versus vectors, [117,](#_bookmark284) [146](#_bookmark354)–[147](#_bookmark357)

**Artiﬁcial Intelligence (AI),** [**203**](#_bookmark488)

**ASCII (American Standard Code for Information Interchange),** [**22,**](#_bookmark76) [**393**](#_bookmark821)**–**[**396**](#_bookmark822)

**askNumber() function**

assigning default arguments to parameters, [173](#_bookmark412)

Mad Lib game, [182](#_bookmark436) Tic-Tac-Toe game, [209](#_bookmark500)

**askText()function, Mad Lib game,** [**182**](#_bookmark436)

**askYesNo1() function**

Mad Lib game, [157](#_bookmark377)–[158](#_bookmark379) Tic-Tac-Toe game, [208](#_bookmark497)–[209](#_bookmark500)

**assembly language,** [**2**](#_bookmark19) **assigning values**

to Boolean variables, [22](#_bookmark76)

to character variables, [21](#_bookmark74)–[22](#_bookmark76) to ﬂoating point variables, [21](#_bookmark74) to integer variables, [21](#_bookmark74)

**assignment operator (=),** [**21,**](#_bookmark74) [**44,**](#_bookmark129)

[**313**](#_bookmark697)**–**[**315,**](#_bookmark699) [**346**](#_bookmark755)**–**[**347**](#_bookmark758)

**assignment statement,** [**70**](#_bookmark185)

**associative container,** [**138**](#_bookmark330)

##### B

**badSwap() function,** [**193**](#_bookmark462)**–**[**194,**](#_bookmark465)

[**236**](#_bookmark553)**–**[**237**](#_bookmark556)

**base class**

constructors, [343](#_bookmark747)–[344,](#_bookmark751) [379](#_bookmark801)

destructors, [379](#_bookmark800)

member functions, [340](#_bookmark741)–[346](#_bookmark755) base class constructors,

[343](#_bookmark746)–[344](#_bookmark750)

calling, [345](#_bookmark753)–[346](#_bookmark755)

declaring virtual, [344](#_bookmark750) Overriding Boss program,

[341](#_bookmark745)–[343](#_bookmark746)

overriding virtual, [344](#_bookmark750)–[345](#_bookmark753)

pointers, [350](#_bookmark761)–[351](#_bookmark762)

**begin()vector member function,** [**127,**](#_bookmark306) [**135**](#_bookmark323)

**Blackjack game,** [**356**](#_bookmark772)**–**[**379**](#_bookmark800)

Card class, [361](#_bookmark778)–[363](#_bookmark780)

Deck class, [370](#_bookmark790)–[373](#_bookmark793)

designing classes, [356](#_bookmark772)–[360](#_bookmark776)

Game class, [373](#_bookmark793)–[376](#_bookmark795) GenericPlayer class, [366](#_bookmark783)–[367](#_bookmark785) Hand class, [363](#_bookmark780)–[366](#_bookmark783)

House class, [369](#_bookmark788)–[370](#_bookmark790)

main() function, [376](#_bookmark795)–[377](#_bookmark797) overloading the operator()

function, [377](#_bookmark797)–[379](#_bookmark800) planning game logic, [360](#_bookmark776)–[361](#_bookmark778) Player class, [368](#_bookmark787)–[369](#_bookmark788)

**block,** [**8,**](#_bookmark35) [**37**](#_bookmark112)

**body of function,** [**8**](#_bookmark35)

**bool fullScreen parameter,** [**171**](#_bookmark409)

**bool IsBusted() const member, BlackJack GenericPlayer class,** [**359**](#_bookmark775)

**bool m\_IsFaceUp member,**

**BlackJack Card class,** [**359**](#_bookmark775)

**bool type,** [**18,**](#_bookmark65) [**19*tbl***](#_bookmark68)**,** [**37**](#_bookmark112)**–**[**38**](#_bookmark114)

**bool variables,** [**39,**](#_bookmark117) [**78**](#_bookmark203)

**Boole, George,** [**38**](#_bookmark114)

**Boolean variables,** [**22**](#_bookmark76)

**bounds checking,** [**93,**](#_bookmark230)

[**100**](#_bookmark245)**–**[**101,**](#_bookmark248) [**111**](#_bookmark271)

**branching,** [**40**](#_bookmark120)**–**[**54**](#_bookmark150)

if statement, [40](#_bookmark120)–[51](#_bookmark144)

with else clauses, [45](#_bookmark131)–[51](#_bookmark144) nesting, [44](#_bookmark129)–[45](#_bookmark131)

relational operators, [44](#_bookmark129) Score Rater program, [41](#_bookmark123)–[42](#_bookmark124) true and false values,

[42](#_bookmark124)–[43](#_bookmark126)

switch statement, [51](#_bookmark144)–[54](#_bookmark150) creating ways to branch, [54](#_bookmark150) Menu Chooser program,

[52](#_bookmark146)–[53](#_bookmark147)

**break statement,** [**79,**](#_bookmark204) [**111**](#_bookmark271)

exiting loop, [60](#_bookmark164)

Finicky Counter program, [58](#_bookmark159)–[60](#_bookmark164)

loops, [77](#_bookmark201)

switch statement and, [51](#_bookmark144) understanding when to use, [61](#_bookmark169) while (true) loop, [60](#_bookmark164)

**busting, BlackJack,** [**356**](#_bookmark772)

##### C

**C programming language,** [**36**](#_bookmark109) **C**þþ **program,** [**5**](#_bookmark25)**–**[**10**](#_bookmark41)

commenting code, [7](#_bookmark30) creating ﬁrst program with,

[383](#_bookmark809)–[388](#_bookmark813)

Index 401

displaying text through standard output, [8](#_bookmark35)–[9](#_bookmark38)

Game Over program, [5](#_bookmark25)–[6](#_bookmark28) including other ﬁles, [7](#_bookmark31)–[8](#_bookmark36) main() function, [8,](#_bookmark35) [10](#_bookmark41)

terminating statements, [9](#_bookmark38)

whitespace, [7](#_bookmark30)

**calling**

base class member functions, [345](#_bookmark753)–[346](#_bookmark755)

constructors, [263](#_bookmark609)–[264](#_bookmark613)

end()vector member function, [127](#_bookmark306)–[128](#_bookmark307)

functions, [70,](#_bookmark185) [154](#_bookmark370)

inlined functions, [179](#_bookmark428)–[180](#_bookmark431)

member functions, [260](#_bookmark603)

overloaded functions, [177](#_bookmark424) static member functions,

[273](#_bookmark629)–[274](#_bookmark633)

**camel case,** [**20**](#_bookmark71)

**capacity()member function,** [**136**](#_bookmark326)**–**[**137**](#_bookmark328)

**Card class, BlackJack,** [**357,**](#_bookmark774)[**359,**](#_bookmark775) [**361**](#_bookmark778)**–**[**363**](#_bookmark780)

**case sensitivity,** [**20,**](#_bookmark71) [**38,**](#_bookmark114) [**258**](#_bookmark595)

**cctype ﬁle,** [**142**](#_bookmark343)

**change\_global() function,** [**169**](#_bookmark402)

**char type, 19*tbl*,** [**112**](#_bookmark273)

**char value,** [**158**](#_bookmark379)

**character variables,** [**21**](#_bookmark74)**–**[**22**](#_bookmark76)

**cin object,** [**36**](#_bookmark109)

**classes,** [**255**](#_bookmark589)**–**[**283**](#_bookmark651)

*See also* advanced classes constructors, [260](#_bookmark603)–[264](#_bookmark613)

calling automatically, [263](#_bookmark609)–[264](#_bookmark613)

Constructor Critter program, [261](#_bookmark606)–[262](#_bookmark607)

declaring and deﬁning, [262](#_bookmark607)–[263](#_bookmark609)

Critter Caretaker game, [274](#_bookmark633)–[281](#_bookmark646)

Critter class, [277](#_bookmark640)–[280](#_bookmark644)

main() function, [280](#_bookmark644)–[281](#_bookmark646)

planning game, [275](#_bookmark635)–[276](#_bookmark636)

planning pseudocode, [276](#_bookmark637)

deﬁned, [282](#_bookmark649)

deﬁning new types, [255](#_bookmark589)–[260](#_bookmark603) accessing data members,

[259](#_bookmark599)–[260](#_bookmark603)

calling member functions, [260](#_bookmark603)

declaring data members, [258](#_bookmark595) declaring member

functions, [258](#_bookmark595)

deﬁning member functions, [258](#_bookmark595)–[259](#_bookmark599)

instatiating objects, [259](#_bookmark599) Simple Critter program,

[256](#_bookmark592)–[257](#_bookmark593)

setting member access levels, [264](#_bookmark613)–[269](#_bookmark620)

deﬁning accessor member functions, [267](#_bookmark616)–[268](#_bookmark618) deﬁning constant member functions, [268](#_bookmark618)–[269](#_bookmark620)

Private Critter program, [264](#_bookmark613)–[266](#_bookmark614)

specifying public and private, [266](#_bookmark614)–[267](#_bookmark616)

static data members accessing, [272](#_bookmark626)–[273](#_bookmark629)

declaring, [272](#_bookmark626)

initializing, [272](#_bookmark626)

Static Critter program, [270](#_bookmark623)–[271](#_bookmark624)

static member functions calling, [273](#_bookmark629)–[274](#_bookmark633)

declaring, [273](#_bookmark629)

deﬁning, [273](#_bookmark630)

Static Critter program, [270](#_bookmark623)–[271](#_bookmark624)

**clear()member function,** [**122**](#_bookmark296)

**code proﬁling,** [**185**](#_bookmark444)**–**[**186**](#_bookmark447) **combined assignment operators,**

[**26**](#_bookmark85)**–**[**27**](#_bookmark89)

**commenting code,** [**7**](#_bookmark30)

**comments,** [**7,**](#_bookmark30) [**37**](#_bookmark112)

**compile errors,** [**4**](#_bookmark23)

**compiler, deﬁned,** [**2**](#_bookmark19)

**compiler warning,** [**37**](#_bookmark112)

**computerMove()function, Tic-Tac-Toe game,** [**213**](#_bookmark510)**–**[**216**](#_bookmark512)

**concatenating string objects,** [**92**](#_bookmark227)

**const keyword,** [**170,**](#_bookmark406) [**234,**](#_bookmark550)

[**249,**](#_bookmark581) [**251,**](#_bookmark585) [**269**](#_bookmark620)

**constant iterator,** [**126**](#_bookmark304) **constant member functions,**

[**268**](#_bookmark618)**–**[**269**](#_bookmark620)

**constant pointer,** [**246**](#_bookmark574)**–**[**247,**](#_bookmark576) [**249**](#_bookmark581)

**constant reference,** [**197**](#_bookmark472)**–**[**198,**](#_bookmark475) [**218**](#_bookmark517)

**constants,** [**29**](#_bookmark93)**–**[**32**](#_bookmark99)

deﬁned, [36](#_bookmark110)

enumerations, [31](#_bookmark97)–[32](#_bookmark99) Game Stats [3.0](#_bookmark14) program,

[29](#_bookmark93)–[30](#_bookmark95)

global, [170](#_bookmark406)

overview, [31](#_bookmark97)

pointers and, [231](#_bookmark545)–[234](#_bookmark550)

**Constructor Critter program,** [**261**](#_bookmark606)**–**[**262**](#_bookmark607)

**constructors,** [**260**](#_bookmark603)**–**[**264**](#_bookmark613)

base class, [379](#_bookmark800)

calling automatically, [263](#_bookmark609)–[264](#_bookmark613) Constructor Critter program,

[261](#_bookmark606)–[262](#_bookmark607)

declaring and deﬁning, [262](#_bookmark607)–[263](#_bookmark609)

deﬁned, [282](#_bookmark649)

instantiating objects from derived class, [336](#_bookmark730)

**containers**

adaptors, [147](#_bookmark357)

associative, [138](#_bookmark330)

data members, [291](#_bookmark663)–[292](#_bookmark665)

deﬁned, [145](#_bookmark351)

sequential, [138](#_bookmark330)–[139](#_bookmark333)

**continue statement,** [**59,**](#_bookmark162) [**61,**](#_bookmark169)

[**77,**](#_bookmark201) [**111**](#_bookmark271)

**copy constructors,** [**326**](#_bookmark718)

declaring, [309](#_bookmark693)–[313](#_bookmark697)

deﬁning, [309](#_bookmark694)–[313](#_bookmark698)

using in derived classes, [346](#_bookmark755)–[347](#_bookmark758)

**Counter program,** [**82**](#_bookmark211)**–**[**84**](#_bookmark212)

**cout object,** [**8**](#_bookmark35)**–**[**9,**](#_bookmark38) [**36**](#_bookmark109)

**Critter Caretaker game,** [**274**](#_bookmark633)**–**[**281**](#_bookmark646)

Critter class, [277](#_bookmark640)–[280](#_bookmark644)

class constructor, [277](#_bookmark640)–[278](#_bookmark642)

class deﬁnition, [277](#_bookmark641)

Eat() member function, [279](#_bookmark643)–[280](#_bookmark644)

GetMood() member function, [278](#_bookmark642) PassTime() member

function, [278](#_bookmark642)

Play() member function, [280](#_bookmark644)

Talk() member function, [279](#_bookmark643)

main() function, [280](#_bookmark644)–[281](#_bookmark646)

planning game, [275](#_bookmark635)–[276](#_bookmark636) planning pseudocode,

[276](#_bookmark637)

**Critter Farm program,** [**288**](#_bookmark660)**–**[**290**](#_bookmark661)

**cstdlib ﬁle,** [**78**](#_bookmark203)

**cstring ﬁle,** [**102**](#_bookmark250)

**C-style comments,** [**7**](#_bookmark30)

**C-style strings,** [**101**](#_bookmark248)**–**[**102**](#_bookmark250) **curly braces**

functions, [8](#_bookmark35)

nested scopes, [165](#_bookmark392)

scopes, [163](#_bookmark388)

402 Index

##### D

**dangling pointers,** [**241,**](#_bookmark563) [**250**](#_bookmark584)**–**[**251,**](#_bookmark585)

[**301,**](#_bookmark682) [**325**](#_bookmark715)

**data members,** [**87,**](#_bookmark219) [**110**](#_bookmark268)

accessing, [259](#_bookmark599)–[260](#_bookmark603)

container, [291](#_bookmark663)–[292](#_bookmark665)

declaring, [258](#_bookmark595)

heap and, [303](#_bookmark685)–[315](#_bookmark699)

declaring and deﬁning copy constructors, [309](#_bookmark693)–[313](#_bookmark697)

declaring and deﬁning destructors, [308](#_bookmark690)–[309](#_bookmark693)

declaring data members that point to values on heap, [307](#_bookmark688)–[308](#_bookmark690)

Heap Data Member program, [303](#_bookmark685)–[307](#_bookmark688)

overloading assignment operator, [313](#_bookmark697)–[315](#_bookmark699)

object, [290](#_bookmark661)–[291](#_bookmark663)

**Deck class, BlackJack,** [**357,**](#_bookmark774) [**360,**](#_bookmark776)

[**370**](#_bookmark790)**–**[**373**](#_bookmark793)

**Deck m\_Deck member, BlackJack**

**Game class,** [**360**](#_bookmark776) **declaring**

constructors, [262](#_bookmark607)–[263](#_bookmark609)

copy constructors, [309](#_bookmark693)–[313](#_bookmark697)

data members, [258,](#_bookmark595) [307](#_bookmark688)–[308](#_bookmark690)

destructors, [308](#_bookmark690)–[309](#_bookmark693)

functions, [153,](#_bookmark368) [183](#_bookmark439)

iterators, [125](#_bookmark302)–[126](#_bookmark304)

member functions, [258](#_bookmark595)

multidimensional arrays, [105](#_bookmark255)

pointers, [226](#_bookmark529)–[227,](#_bookmark532) [249](#_bookmark582)

references, [189](#_bookmark454)

static data members, [272](#_bookmark626) static member functions, [273](#_bookmark629) variables, [19](#_bookmark68)–[20,](#_bookmark72) [165](#_bookmark393)

vectors, [119](#_bookmark286)–[120](#_bookmark289)

**deep copy,** [**310,**](#_bookmark695) [**326**](#_bookmark718) **default arguments,** [**171**](#_bookmark409)**–**[**174,**](#_bookmark415) [**184**](#_bookmark441)

assigning to parameters, [173](#_bookmark412)–[174](#_bookmark415)

Give Me a Number program, [171](#_bookmark409)–[172](#_bookmark410)

overriding, [174](#_bookmark415)

specifying, [173](#_bookmark412)

**default assignment operator,** [**313,**](#_bookmark697) [**326**](#_bookmark718)

**default constructor,** [**262**](#_bookmark607)**–**[**263,**](#_bookmark609) [**284**](#_bookmark653)

**default copy constructor,** [**309,**](#_bookmark693) [**326**](#_bookmark718)

**default destructor,** [**308**](#_bookmark690)

**default keyword,** [**77**](#_bookmark201)

**default statement,** [**51**](#_bookmark144)

**deﬁning**

accessor member functions, [267](#_bookmark616)–[268](#_bookmark618)

constant member functions, [268](#_bookmark618)–[269](#_bookmark620)

constructors, [262](#_bookmark607)–[263](#_bookmark609)

copy constructors, [309](#_bookmark693)–[313](#_bookmark697)

destructors, [308](#_bookmark690)–[309](#_bookmark693)

enumerations, [31](#_bookmark97)–[32](#_bookmark99)

functions, [154](#_bookmark370)

member functions, [258](#_bookmark596)–[259,](#_bookmark600) [282](#_bookmark649) new names for types, [23](#_bookmark79)

new types, [255](#_bookmark589)–[260](#_bookmark603) accessing data members,

[259](#_bookmark599)–[260](#_bookmark603)

calling member functions, [260](#_bookmark603)

declaring data members, [258](#_bookmark595) declaring member

functions, [258](#_bookmark595)

deﬁning member functions, [258](#_bookmark595)–[259](#_bookmark599)

instatiating objects, [259](#_bookmark599) Simple Critter program,

[256](#_bookmark592)–[257](#_bookmark593)

static member functions, [273](#_bookmark629) variables inside switch

statements, [166](#_bookmark395)

virtual destructors, [351](#_bookmark762)–[352](#_bookmark764)

**delete operator,** [**300**](#_bookmark680)**–**[**301,**](#_bookmark682) [**325**](#_bookmark715) **deque STL container,** [**139**](#_bookmark333) **dereference operator (\*),** [**128,**](#_bookmark307) [**146,**](#_bookmark354)

[**230**](#_bookmark542)

**dereferencing**

iterators, [128](#_bookmark307)

pointers, [228](#_bookmark535)–[229,](#_bookmark538) [249](#_bookmark582)

**derived classes,** [**379**](#_bookmark800)

from base classes, [335](#_bookmark728)–[336](#_bookmark730) deriving from abstract classes,

[355](#_bookmark770)–[356](#_bookmark772)

instantiating objects from, [336](#_bookmark730)–[337](#_bookmark733)

using access modiﬁers when deriving, [340](#_bookmark741)

using base class pointers to objects, [350](#_bookmark761)–[351](#_bookmark762)

using overloaded assignment operators and copy constructors in, [346](#_bookmark755)–[347](#_bookmark758)

**Designers Network Program,** [**62**](#_bookmark172)**–**[**65**](#_bookmark174)

**destructors,** [**326**](#_bookmark718)

base class, [379](#_bookmark800)

declaring, [308](#_bookmark690)–[309](#_bookmark693)

deﬁning, [308](#_bookmark691)–[309](#_bookmark694)

instantiating objects from derived class, [336](#_bookmark730)

virtual, [347](#_bookmark758)

when to make virtual, [381](#_bookmark804) **Die Roller program,** [**68**](#_bookmark181)**–**[**69**](#_bookmark183) **display() function**

constant referencees and, [197](#_bookmark472) passing and returning arrays,

[247](#_bookmark576)–[248](#_bookmark578)

**displayBoard()function,**

**Tic-Tac-Toe game,** [**210**](#_bookmark503)**–**[**211**](#_bookmark506)

**do loops,** [**56**](#_bookmark155)**–**[**58,**](#_bookmark159) [**77**](#_bookmark201)

logical NOT operator, [66](#_bookmark176) overview, [57](#_bookmark157)–[58](#_bookmark159)

Play Again 2.0 program, [56](#_bookmark155)–[57](#_bookmark157)

**dot operator,** [**230**](#_bookmark542) **double type,** [**18,**](#_bookmark65) [**19*tbl***](#_bookmark68)**double-ended queue,** [**148**](#_bookmark358)

**dynamic array,** [**116**](#_bookmark282) **dynamic memory**

allocating, [296](#_bookmark673)–[303](#_bookmark685)

avoiding memory leaks, [301–](#_bookmark682) [303](#_bookmark685)

delete operator, [300](#_bookmark680)–[301](#_bookmark682)

Heap program, [297](#_bookmark677)–[299](#_bookmark679)

new operator, [299](#_bookmark679)–[300](#_bookmark680) data members and heap,

[303](#_bookmark685)–[315](#_bookmark699)

copy constructors, [309](#_bookmark693)–[313](#_bookmark697)

declaring, [307](#_bookmark688)–[308](#_bookmark690)

destructors, [308](#_bookmark690)–[309](#_bookmark693) Heap Data Member

program, [303](#_bookmark685)–[307](#_bookmark688) overloading assignment op-

erator, [313](#_bookmark697)–[315](#_bookmark699)

##### E

**early binding,** [**351**](#_bookmark762)

**Eat() member function, Critter**

**class,** [**279**](#_bookmark643)**–**[**280**](#_bookmark644)

**editor, deﬁned,** [**2**](#_bookmark19)

**else clause, if statement,** [**45**](#_bookmark131)**–**[**51,**](#_bookmark144) [**77**](#_bookmark201)

creating ways to branch, [47](#_bookmark135)–[48](#_bookmark138) if statements with, [48](#_bookmark138)–[51](#_bookmark144) Score Rater 2.0 program, [46](#_bookmark134)–[47](#_bookmark135)

**empty statements, for loops,** [**85**](#_bookmark214)

**empty()member function,** [**96,**](#_bookmark238) [**122**](#_bookmark296)

**encapsulation,** [**160,**](#_bookmark383) [**184,**](#_bookmark441) [**284**](#_bookmark653)

**end() member function**

calling, [127](#_bookmark306)–[128](#_bookmark307)

string objects, [135](#_bookmark323)

Index 403

**enumerations,** [**31**](#_bookmark97)**–**[**32,**](#_bookmark99) [**36**](#_bookmark109)

**enumerators,** [**31**](#_bookmark97)

**equal to (==) operator, 40*tbl*,** [**44**](#_bookmark129)

**erase()member function,** [**95,**](#_bookmark235)

[**130**](#_bookmark312)**–**[**131**](#_bookmark315)

**errors, 4,** [**35**](#_bookmark106)

**escape sequences,** [**23,**](#_bookmark79) [**397**](#_bookmark824)

**executable ﬁle,** [**2**](#_bookmark19)**–**[**4**](#_bookmark23)

**exit statement,** [**85**](#_bookmark214)

**Expensive Calculator program,**[**13**](#_bookmark52)**–**[**14**](#_bookmark54)

**expression,** [**14**](#_bookmark54)

##### F

**false values**

interpreting, [43](#_bookmark126)

testing, [42](#_bookmark124)–[43](#_bookmark126)

**ﬁnd() algorithm,** [**134**](#_bookmark320)

**ﬁnd() member function,** [**94**](#_bookmark233)**–**[**95**](#_bookmark235) **Finicky Counter program,**

[**58**](#_bookmark159)**–**[**60**](#_bookmark164)

**ﬂoat type,** [**19*tbl***](#_bookmark68)

**ﬂoating points,** [**14,**](#_bookmark54) [**21**](#_bookmark74)

**for loops,** [**81**](#_bookmark208)**–**[**87**](#_bookmark219)

Counter program, [82](#_bookmark211)–[84](#_bookmark212)

counting with, [84](#_bookmark212)–[85](#_bookmark214) deﬁning variables inside, [166](#_bookmark395) nesting, [86](#_bookmark217)–[87](#_bookmark219)

using empty statements in, [85](#_bookmark214) **FORTRAN language,** [**112**](#_bookmark273) **forward slash (/),** [**14**](#_bookmark54)

**free store,** [**296,**](#_bookmark673) [**325.**](#_bookmark715) ***See also* heap freeing memory,** [**300**](#_bookmark680)

**Friend Critter program,** [**292**](#_bookmark665)**–**[**294**](#_bookmark668)

**friend functions,** [**325**](#_bookmark715)

creating, [295](#_bookmark670)

Friend Critter program, [292](#_bookmark665)–[294](#_bookmark668)

**function body,** [**154**](#_bookmark370)

**function header,** [**154**](#_bookmark370)

**function inlining,** [**184**](#_bookmark441)

**functions, 8,** [**35,**](#_bookmark106) [**151**](#_bookmark363)**–**[**186**](#_bookmark447)

abstraction, [155](#_bookmark374) accepting values into

parameters, [158](#_bookmark379)–[160](#_bookmark383)

calling, [70,](#_bookmark185) [154](#_bookmark370) calling with default

arguments, [173](#_bookmark412)

declaring, [153,](#_bookmark368) [183](#_bookmark439)

default arguments, [171](#_bookmark409)–[174](#_bookmark415) assigning to parameters,

[173](#_bookmark412)–[174](#_bookmark415)

Give Me a Number program, [171](#_bookmark409)–[172](#_bookmark410)

overriding, [174](#_bookmark415)

specifying, [173](#_bookmark412)

deﬁning, [154](#_bookmark371)

encapsulation, [160](#_bookmark383)

global constants, [170](#_bookmark406)

global variables, [166](#_bookmark395)–[170](#_bookmark406)

accessing, [168](#_bookmark399)

altering, [169](#_bookmark402)

declaring, [168](#_bookmark399)

Global Reach program, [166](#_bookmark395)–[168](#_bookmark399)

hiding, [169](#_bookmark402) minimizing use of, [170](#_bookmark406)

inlining, [177](#_bookmark424)–[180](#_bookmark431)

calling, [179](#_bookmark428)–[180](#_bookmark431)

specifying, [179](#_bookmark428)

Taking Damage program, [177](#_bookmark424)–[179](#_bookmark428)

Instructions program, [152](#_bookmark366)–[153](#_bookmark368) Mad Lib game, [180](#_bookmark431)–[183](#_bookmark439)

askNumber()function, [182](#_bookmark436)

askText()function, [182](#_bookmark436)

main()function, [181](#_bookmark433)–[182](#_bookmark436)

setting up, [181](#_bookmark433)

tellStory()function, [183](#_bookmark439)

overloading, [174](#_bookmark416)–[177,](#_bookmark425) [184](#_bookmark442)

calling, [177](#_bookmark424)

creating, [176](#_bookmark421)

Triple program, [174](#_bookmark415)–[176](#_bookmark421)

prototype, [153,](#_bookmark368) [159](#_bookmark381)

return values, [157](#_bookmark377)–[158](#_bookmark379)

return statement, [158](#_bookmark379) specifying return type, [157](#_bookmark377) using, [158](#_bookmark379)

scopes, [161](#_bookmark385)–[166](#_bookmark395)

nested, [165](#_bookmark392)–[166](#_bookmark395)

Scoping program, [161](#_bookmark385)–[163](#_bookmark388)

separate, [163](#_bookmark388)–[164](#_bookmark390)

software reuse, [161](#_bookmark385)

Tic-Tac-Toe game, [206***tbl***](#_bookmark493)Yes or No program, [155](#_bookmark374)–[157](#_bookmark377)

**fundamental types,** [**18**](#_bookmark65)

##### G

**Game class, BlackJack,** [**357,**](#_bookmark774) [**360,**](#_bookmark776)

[**373**](#_bookmark793)**–**[**376**](#_bookmark795)

**Game Lobby program,** [**315**](#_bookmark699)**–**[**325**](#_bookmark715)

Lobby class, [318](#_bookmark703)–[320](#_bookmark707)

Lobby::AddPlayer() member function, [320](#_bookmark707)–[322](#_bookmark709)

Lobby::Clear() member function, [322](#_bookmark709)–[323](#_bookmark711)

Lobby::RemovePlayer()

member function, [322](#_bookmark709)

main() function, [324](#_bookmark713)–[325](#_bookmark715)

operator() member function, [323](#_bookmark711)–[324](#_bookmark713)

Player class, [316](#_bookmark701)–[318](#_bookmark703)

**game loop,** [**72**](#_bookmark190)**–**[**73,**](#_bookmark191) [**77**](#_bookmark201)

**Game Over program,** [**5**](#_bookmark25)**–**[**6,**](#_bookmark28) [**10**](#_bookmark41)**–**[**12**](#_bookmark48)

**Game Stats 2.0 program,** [**24**](#_bookmark83)**–**[**26**](#_bookmark85)

**Game Stats 3.0 program,** [**29**](#_bookmark93)**–**[**30**](#_bookmark95) **Game Stats program,** [**16**](#_bookmark61)**–**[**17**](#_bookmark62) **game testers,** [**4**](#_bookmark23)

**generating random numbers,** [**68**](#_bookmark181)**–**[**72**](#_bookmark190)

calculating number within range, [71](#_bookmark187)–[72](#_bookmark190)

Die Roller program, [68](#_bookmark181)–[69](#_bookmark183)

rand()function, [69](#_bookmark183)–[70](#_bookmark185)

seeding, [70](#_bookmark185)–[71](#_bookmark187)

**generic statement,** [**40**](#_bookmark120)

**GenericPlayer class, BlackJack,** [**357,**](#_bookmark774) [**359,**](#_bookmark775) [**366**](#_bookmark783)**–**[**367**](#_bookmark785)

**GetMood()member function,**

**Critter class,** [**278**](#_bookmark642)

**Give Me a Number program,** [**171**](#_bookmark409)**–**[**172**](#_bookmark410)

**global constants,** [**170,**](#_bookmark406) [**184**](#_bookmark441)

**global variables,** [**166**](#_bookmark395)**–**[**170,**](#_bookmark406) [**184**](#_bookmark441)

accessing, [168](#_bookmark399)

altering, [169](#_bookmark402)

declaring, [168](#_bookmark399)

Global Reach program, [166](#_bookmark395)–[168](#_bookmark399)

hiding, [169](#_bookmark402) minimizing use of, [170](#_bookmark406)

**goodSwap() header,** [**192,**](#_bookmark461) [**194,**](#_bookmark465) [**237**](#_bookmark556) **greater than (>) operator,** [**40*tbl***](#_bookmark119)**greater than or equal to (>=)**

**operator,** [**40*tbl***](#_bookmark119)

**Guess My Number game,** [**73**](#_bookmark191)**–**[**76**](#_bookmark198)applying game loop, [74](#_bookmark194) creating, [76](#_bookmark198)

ending, [76](#_bookmark198)

setting up, [74](#_bookmark194)–[75](#_bookmark196)

##### H

**Hand class, BlackJack,** [**357,**](#_bookmark774) [**359,**](#_bookmark775)

[**363**](#_bookmark780)**–**[**366**](#_bookmark783)

**Hangman game,** [**141**](#_bookmark340)**–**[**145**](#_bookmark352)

ending, [145](#_bookmark352)

entering main loop, [143](#_bookmark346)–[144](#_bookmark349) getting player's guess,

[144](#_bookmark349)–[145](#_bookmark352)

initializing variables and constants, [143](#_bookmark346)

planning, [141](#_bookmark340)–[142](#_bookmark343)

setting up, [142](#_bookmark343)

404 Index

**has-a relationships,** [**290**](#_bookmark661)

**header ﬁle,** [**8**](#_bookmark35) **heap**

data members and, [303](#_bookmark685)–[315](#_bookmark699) declaring and deﬁning copy

constructors, [309](#_bookmark693)–[313](#_bookmark697) declaring and deﬁning

destructors, [308](#_bookmark690)–[309](#_bookmark693) declaring to values on heap,

[307](#_bookmark688)–[308](#_bookmark690)

Heap Data Member program, [303](#_bookmark685)–[307](#_bookmark688)

overloading assignment operator, [313](#_bookmark697)–[315](#_bookmark699)

memory, [296,](#_bookmark673) [325](#_bookmark715)

**Heap Data Member program,** [**303**](#_bookmark685)**–**[**307**](#_bookmark688)

**Heap program,** [**297**](#_bookmark677)**–**[**299**](#_bookmark679) **Hero's Inventory program,**

[**96**](#_bookmark238)**–**[**98**](#_bookmark241)

version 2.0, [117](#_bookmark284)–[119](#_bookmark286)

version 3.0, [123](#_bookmark301)–[125](#_bookmark302)

**hide\_global() function,** [**169**](#_bookmark402) **hiding**

global variables, [169](#_bookmark402)

variables, [185](#_bookmark444)

**high parameter, askNumber()**

**function,** [**173**](#_bookmark412)

**High Scores program,** [**131**](#_bookmark315)**–**[**133**](#_bookmark317)

**House class, BlackJack,** [**357,**](#_bookmark774) [**360,**](#_bookmark776)

[**369**](#_bookmark788)**–**[**370**](#_bookmark790)

**House m\_House member,**

**BlackJack Game class,** [**360**](#_bookmark776)

**humanMove()function,**

**Tic-Tac-Toe game,** [**213**](#_bookmark510)

**humanPiece()function, Tic-Tac-Toe game,** [**209**](#_bookmark500)**–**[**210**](#_bookmark503)

##### I

**IDE (Integrated Development Environment),** [**3**](#_bookmark21)**–**[**4**](#_bookmark23)

**identiﬁer,** [**20,**](#_bookmark71) [**31**](#_bookmark97)

**if statements,** [**39**](#_bookmark117)**–**[**45**](#_bookmark131)

deﬁning variables inside, [166](#_bookmark395) with else clauses, [48](#_bookmark138)–[51](#_bookmark144)

creating sequence of, [50](#_bookmark141)–[51](#_bookmark144) Score Rater 3.0 program,

[49](#_bookmark140)–[50](#_bookmark141)

nesting, [44](#_bookmark129)–[45](#_bookmark131)

relational operators, [44](#_bookmark129) Score Rater program, [41](#_bookmark123)–[42](#_bookmark124) true and false values

interpreting, [43](#_bookmark126)

testing, [42](#_bookmark124)–[43](#_bookmark126)

**increment operator (**þþ**),** [**27**](#_bookmark89)**–**[**28**](#_bookmark90)

**indexing**

arrays, [99–](#_bookmark243)[100](#_bookmark245) multidimensional arrays,

[105](#_bookmark255)–[106](#_bookmark258)

string objects, [93](#_bookmark230)

vectors, [121](#_bookmark292)

**inﬁnite loop,** [**58,**](#_bookmark159) [**78**](#_bookmark203)**–**[**79**](#_bookmark204)

**inheritance,** [**331**](#_bookmark723)**–**[**346,**](#_bookmark755) [**356**](#_bookmark772)**–**[**380**](#_bookmark802)

base class member functions, [340](#_bookmark741)–[346](#_bookmark755)

calling, [345](#_bookmark753)–[346](#_bookmark755) calling base class constructors,

[343](#_bookmark746)–[344](#_bookmark750)

declaring virtual, [344](#_bookmark750) Overriding Boss program,

[341](#_bookmark745)–[343](#_bookmark746)

overriding virtual, [344](#_bookmark750)–[345](#_bookmark753)

Blackjack game, [356](#_bookmark772)–[379](#_bookmark800)

Card class, [361](#_bookmark778)–[363](#_bookmark780)

Deck class, [370](#_bookmark790)–[373](#_bookmark793)

designing classes, [356](#_bookmark772)–[360](#_bookmark776)

Game class, [373](#_bookmark793)–[376](#_bookmark795)

GenericPlayer class, [366](#_bookmark783)–[367](#_bookmark785)

Hand class, [363](#_bookmark780)–[366](#_bookmark783)

House class, [369](#_bookmark788)–[370](#_bookmark790)

main() function, [376](#_bookmark795)–[377](#_bookmark797) overloading the

operator()

function, [377](#_bookmark797)–[379](#_bookmark800) planning game logic,

[360](#_bookmark776)–[361](#_bookmark778)

Player class, [368](#_bookmark787)–[369](#_bookmark788) controlling access under,

[337](#_bookmark733)–[340](#_bookmark741)

access modiﬁers, [339](#_bookmark738)–[340](#_bookmark742) Simple Boss 2.0 program,

[338](#_bookmark736)–[339](#_bookmark737)

deriving from base classes, [335](#_bookmark728)–[336](#_bookmark730)

inherited members, [337](#_bookmark733) instantiating objects from

derived classes, [336](#_bookmark730)–[337](#_bookmark733) overloaded assignment

operators and copy constructors, [346](#_bookmark755)–[347](#_bookmark758)

Simple Boss program, [333](#_bookmark727)–[335](#_bookmark728)

**initialization statement, for**

**loop,** [**82**](#_bookmark211)

**initializer list,** [**99**](#_bookmark243) **initializing**

constant pointer, [231](#_bookmark545)

pointers, [227,](#_bookmark533) [250](#_bookmark584)

references, [189](#_bookmark454)

static data members, [272](#_bookmark626) variables, [22](#_bookmark76)

**inlining functions,** [**177**](#_bookmark424)**–**[**180,**](#_bookmark431) [**268,**](#_bookmark618)

[**282**](#_bookmark648)

calling inlined functions, [179](#_bookmark428)–[180](#_bookmark431)

specifying functions for, [179](#_bookmark428) Taking Damage program,

[177](#_bookmark424)–[179](#_bookmark428)

**inner loop,** [**86**](#_bookmark217)**–**[**87**](#_bookmark219)

**insert()vector member function,** [**130**](#_bookmark312)

**instances, class,** [**259**](#_bookmark599)

**instantiating objects,** [**259,**](#_bookmark599) [**263**](#_bookmark609)

**instructions() function**

Mad Lib game, [154](#_bookmark370) Tic-Tac-Toe game, [208](#_bookmark497)

**Instructions program,** [**152**](#_bookmark366)**–**[**153**](#_bookmark368)

**int type,** [**18,**](#_bookmark65) [**19*tbl*,**](#_bookmark68) [**112**](#_bookmark273)

**int argument,** [**175**](#_bookmark419)

**int GetTotal() const member,**

**BlackJack Hand class,** [**359**](#_bookmark775)

**int GetValue() member,**

**BlackJack Card class,** [**359**](#_bookmark775) **integer division,** [**14**](#_bookmark54)**–**[**15**](#_bookmark57)

**integer variables,** [**21**](#_bookmark74) **integer wrap around,** [**26,**](#_bookmark85)

[**28**](#_bookmark90)**–**[**29**](#_bookmark93)

**integers,** [**14,**](#_bookmark54) [**78**](#_bookmark203) **Integrated Development**

**Environment (IDE),** [**3**](#_bookmark21)**–**[**4**](#_bookmark23)

**intOnHeap() function,** [**300**](#_bookmark680) **Inventory Displayer program,**

[**195**](#_bookmark468)**–**[**196**](#_bookmark469)

**Inventory Pointer program,** [**239**](#_bookmark560)**–**[**240**](#_bookmark561)

**iostream ﬁle,** [**8**](#_bookmark35)**–**[**9,**](#_bookmark38) [**36**](#_bookmark109)

**isLegal()function, Tic-Tac-Toe game,** [**212**](#_bookmark507)**–**[**213**](#_bookmark510)

**ISO standard,** [**5**](#_bookmark25)

**iterating through string objects,** [**93**](#_bookmark230)**–**[**94**](#_bookmark233)

**iterators,** [**123**](#_bookmark301)**–**[**131**](#_bookmark315)

accessing member functions of vector element, [129](#_bookmark310)–[130](#_bookmark312)

changing value of vector element, [128](#_bookmark307)–[129](#_bookmark310)

declaring, [125](#_bookmark302)–[126](#_bookmark304)

deﬁned, [116,](#_bookmark281) [146](#_bookmark355)

erase()vector member function, [130](#_bookmark312)–[131](#_bookmark315)

Hero's Inventory 3.0 program, [123](#_bookmark301)–[125](#_bookmark302)

insert()vector member function, [130](#_bookmark312)

Index 405

looping through vector, [126](#_bookmark304)–[128](#_bookmark307) altering an iterator, [128](#_bookmark307) calling begin()vector

member function, [127](#_bookmark306) calling end()vector

member function, [127](#_bookmark306)–[128](#_bookmark307)

dereferencing an iterator, [128](#_bookmark307)

vector member functions and, [147](#_bookmark357)

##### K

**keywords,** [**20,**](#_bookmark71) [**391**](#_bookmark818)**–**[**392**](#_bookmark819)

##### L

**late binding,** [**351**](#_bookmark762)

**length() member function,** [**92**](#_bookmark227) **less than (***<***) operator,** [**40*tbl***](#_bookmark119)**less than or equal to (***<***=)**

**operator,** [**40*tbl***](#_bookmark119)**link errors,** [**4**](#_bookmark23)

**linked list,** [**318**](#_bookmark703)**–**[**319**](#_bookmark705) **list STL container,** [**139**](#_bookmark333) **literals,** [**37**](#_bookmark112)

**Lobby class, Game Lobby program,** [**318**](#_bookmark703)**–**[**320**](#_bookmark707)

**Lobby::AddPlayer() member function, Game Lobby program,** [**320**](#_bookmark707)**–**[**322**](#_bookmark709)

**Lobby::Clear() member function, Game Lobby program,** [**322**](#_bookmark709)**–**[**323**](#_bookmark711)

**Lobby::RemovePlayer() member function, Game Lobby program,** [**322**](#_bookmark709)

**local variables,** [**163,**](#_bookmark388) [**185**](#_bookmark444)

**logical errors,** [**4**](#_bookmark23)

**logical operators,** [**61**](#_bookmark169)**–**[**68**](#_bookmark181)

Designers Network Program, [62](#_bookmark172)–[65](#_bookmark174)

NOT operator, [66](#_bookmark176)–[67](#_bookmark178)

AND operator, [65](#_bookmark174)

OR operator, [66](#_bookmark176)

order of operations, [67](#_bookmark178)–[68](#_bookmark181) **long double type,** [**19*tbl***](#_bookmark68)**long int type,** [**19*tbl***](#_bookmark68)

**long modiﬁer,** [**18**](#_bookmark65)

**loop body,** [**56**](#_bookmark155) **looping**

break statement exiting loop, [60](#_bookmark164)

Finicky Counter program, [58](#_bookmark159)–[60](#_bookmark164)

understanding when to use, [61](#_bookmark169)

while (true) loop, [60](#_bookmark164)

continue statement

Finicky Counter program, [58](#_bookmark159)–[60](#_bookmark164)

jumping back to top of loop, [61](#_bookmark169)

understanding when to use, [61](#_bookmark169)

while (true) loop, [60](#_bookmark164)

do loops, [56](#_bookmark155)–[58](#_bookmark159)

overview, [57](#_bookmark157)–[58](#_bookmark159)

Play Again 2.0 program, [56](#_bookmark155)–[57](#_bookmark157)

iterators through vector, [126](#_bookmark304)–[128](#_bookmark307)

altering an iterator, [128](#_bookmark307) calling begin()vector

member function, [127](#_bookmark306) calling end()vector

member function, [127](#_bookmark306)–[128](#_bookmark307)

dereferencing an iterator, [128](#_bookmark307)

logical operators, [61](#_bookmark169)–[68](#_bookmark181) Designers Network

Program, [62](#_bookmark172)–[65](#_bookmark174)

NOT operator, [66](#_bookmark176)–[67](#_bookmark178)

AND operator, [65](#_bookmark174)

OR operator, [66](#_bookmark176)

order of operations, [67](#_bookmark178)–[68](#_bookmark181)

while loops, [54](#_bookmark150)–[56](#_bookmark155)

overview, [55](#_bookmark152)–[56](#_bookmark155)

Play Again program, [54](#_bookmark150)–[55](#_bookmark152)

**Lost Fortune game,** [**32**](#_bookmark99)**–**[**35**](#_bookmark106)

getting information from player, [33](#_bookmark102)–[34](#_bookmark104)

setting up, [32](#_bookmark99)–[33](#_bookmark102)

telling story, [34](#_bookmark104)–[35](#_bookmark106)

**low parameter, askNumber()**

**function,** [**173**](#_bookmark412)

##### M

**Mad Lib game,** [**180**](#_bookmark431)**–**[**183**](#_bookmark439)

askNumber()function, [182](#_bookmark436)

askText()function, [182](#_bookmark436)

main()function, [181](#_bookmark433)–[182](#_bookmark436)

setting up, [181](#_bookmark433)

tellStory()function, [183](#_bookmark439)

**main() function,** [**8,**](#_bookmark35) [**181**](#_bookmark433)**–**[**182**](#_bookmark436)

Blackjack game, [376](#_bookmark795)–[377](#_bookmark797) creating string objects, [91](#_bookmark224) Critter Caretaker game,

[280](#_bookmark644)–[281](#_bookmark646)

declaring vectors, [119](#_bookmark286)

deﬁned, [35](#_bookmark107)

Game Lobby program, [324](#_bookmark713)–[325](#_bookmark715)

initializing variables and constants, [143](#_bookmark346)

looping, [55](#_bookmark152)

returning value from, [10](#_bookmark41) Tic-Tac-Toe game, [207](#_bookmark495)–[208](#_bookmark497)

**map STL container,** [**139**](#_bookmark333) **member access levels, class,**

[**264**](#_bookmark613)**–**[**269**](#_bookmark620)

deﬁning accessor member functions, [267](#_bookmark616)–[268](#_bookmark618) deﬁning constant member functions, [268](#_bookmark618)–[269](#_bookmark620)

private, [267](#_bookmark616)

Private Critter program, [264](#_bookmark613)–[266](#_bookmark614)

public, [266](#_bookmark614)

**member access operator,** [**230**](#_bookmark542) **member functions,** [**87,**](#_bookmark219) [**110**](#_bookmark268)

*See also names of speci*ﬁ*c member functions*

calling, [260](#_bookmark603)

declaring, [258](#_bookmark595)

deﬁning, [258](#_bookmark596)–[259,](#_bookmark600) [282](#_bookmark649)

inlining, [282](#_bookmark648)

**member initializers,** [**263,**](#_bookmark609) [**282**](#_bookmark648) **member selection operator (.),** [**88,**](#_bookmark220)

[**259**](#_bookmark599)

**member-wise copying,** [**309,**](#_bookmark693) [**326**](#_bookmark718)

**memory,** [**24,**](#_bookmark83) [**224,**](#_bookmark527) [**248**](#_bookmark578)

**memory leak,** [**301**](#_bookmark682)**–**[**303,**](#_bookmark685) [**326**](#_bookmark718) **Menu Chooser program,**

[**52**](#_bookmark146)**–**[**53**](#_bookmark147)

**modiﬁers, type,** [**18**](#_bookmark65)**–**[**19**](#_bookmark69)

**modulus operator (%),** [**15,**](#_bookmark57) [**72**](#_bookmark190) **multidimensional arrays,**

[**103**](#_bookmark253)**–**[**106,**](#_bookmark258) [**111**](#_bookmark271)

creating, [105](#_bookmark255)

indexing, [105](#_bookmark255)–[106](#_bookmark258)

Tic-Tac-Toe Board program, [103](#_bookmark253)–[105](#_bookmark255)

**multimap STL container,** [**139**](#_bookmark333) **multiple inheritance,** [**380**](#_bookmark802) **multiplication (\*) operator,** [**14**](#_bookmark54) **multiset STL container,** [**139**](#_bookmark333) **mutable data members,** [**284**](#_bookmark653)

##### N

**namespace,** [**9.**](#_bookmark38) ***See also* std**

**namespace**

**naming convention,** [**258**](#_bookmark595)

**naming variables,** [**20**](#_bookmark71)**–**[**21**](#_bookmark74)

406 Index

**nested scopes,** [**165**](#_bookmark392)**–**[**166,**](#_bookmark395) [**185**](#_bookmark444) **nesting**

if statements, [44](#_bookmark129)–[45](#_bookmark131)

for loops, [86](#_bookmark217)–[87](#_bookmark219)

**new operator,** [**299**](#_bookmark679)**–**[**300,**](#_bookmark680) [**325**](#_bookmark715) **New Project dialog box,** [**384**](#_bookmark810)**ﬁ**[**g**](#_bookmark810) **newline (\nl),** [**23**](#_bookmark79)

**\nl (newline),** [**23**](#_bookmark79) **nodes, linked list,** [**319**](#_bookmark705)

**non-constant reference,** [**219**](#_bookmark519) **non-inlined function,**

**calling,** [**179**](#_bookmark428)

**non-virtual member functions, overriding,** [**344,**](#_bookmark750) [**351**](#_bookmark762)

**non-zero value,** [**43,**](#_bookmark126) [**77**](#_bookmark201)

**NOT (!) operator,** [**61*tbl***](#_bookmark168)**,** [**66**](#_bookmark176)**–**[**67,**](#_bookmark178) [**77**](#_bookmark201) **not equal to (!=) operator,** [**40*tbl***](#_bookmark119)**null character,** [**101**](#_bookmark248)

**null pointers,** [**227**](#_bookmark533)**–**[**228,**](#_bookmark536)

[**249,**](#_bookmark581) [**251**](#_bookmark585)

##### O

**object data members,** [**290**](#_bookmark661)**–**[**291**](#_bookmark663) **object ﬁle,** [**2**](#_bookmark19)

**object-oriented programming. *See***

**OOP**

**objects**

***See also*** string objects altering through returning

pointers, [243](#_bookmark568)–[244](#_bookmark571)

deﬁned, [283](#_bookmark652)

instatiating, [259](#_bookmark599)

overview, [87](#_bookmark219)–[89](#_bookmark222)

**OOP (object-oriented programming),** [**2,**](#_bookmark19) [**255,**](#_bookmark589)

[**281,**](#_bookmark646) [**283**](#_bookmark651)

inheritance, [331](#_bookmark723)–[333](#_bookmark727)

polymorphism, [347](#_bookmark758)

**operator overloading,** [**92,**](#_bookmark227)

[**112**](#_bookmark273)**–**[**113,**](#_bookmark274) [**295**](#_bookmark670)**–**[**296,**](#_bookmark673) [**325**](#_bookmark715)

**operator precedence,** [**389**](#_bookmark815)**–**[**390**](#_bookmark816) **operator() member function** Blackjack game, [377](#_bookmark797)–[379](#_bookmark800)

Game Lobby program, [323](#_bookmark711)–[324](#_bookmark713)

**operators. *See speci*ﬁ*c operators***

**opponent()function, Tic-Tac-Toe game,** [**210**](#_bookmark503)

**OR (||) operator,** [**61*tbl***](#_bookmark168)**,** [**66**](#_bookmark176) **order of operations**

arithmetic operators, [15](#_bookmark57)

logical operators, [67](#_bookmark178)–[68](#_bookmark181)

**outer loop,** [**86**](#_bookmark217)**–**[**87**](#_bookmark219) **out-of-scope**

objects, [201,](#_bookmark481) [241](#_bookmark563)

variables, [164,](#_bookmark390) [185](#_bookmark444) **output operator (),** [**9**](#_bookmark38) **output stream,** [**8**](#_bookmark35) **overloading**

assignment operator, [313](#_bookmark697)–[315](#_bookmark699)

functions, [174](#_bookmark415)–[177](#_bookmark424)

calling, [177](#_bookmark424)

creating, [176](#_bookmark421)

Triple program, [174](#_bookmark415)–[176](#_bookmark421)

operators, [92,](#_bookmark227) [112](#_bookmark273)–[113,](#_bookmark274)

[295](#_bookmark671)–[296,](#_bookmark674) [325](#_bookmark716)

versus overriding, [345](#_bookmark753)

**overriding**

base class member functions, [379](#_bookmark800)

default arguments, [174](#_bookmark415)

non-virtual member function, [344,](#_bookmark750) [351](#_bookmark762)

**Overriding Boss program,** [**341**](#_bookmark745)**–**[**343**](#_bookmark746)

##### P

**pAPointer pointer,** [**226**](#_bookmark530)

**parameters,** [**153**](#_bookmark368)

versus arguments, [184](#_bookmark441) assigning default arguments,

[173](#_bookmark412)–[174](#_bookmark415)

functions, [158](#_bookmark379)–[160](#_bookmark383)

**parentheses**

dereferencing iterators, [129](#_bookmark310)

order of operations, [15,](#_bookmark57) [67](#_bookmark178)

**passing pointers,** [**234**](#_bookmark550)**–**[**238**](#_bookmark557)

constant pointer, [237](#_bookmark556)–[238](#_bookmark557) Swap Pointer Version program,

[234](#_bookmark550)–[236](#_bookmark553)

by value, [236](#_bookmark553)–[237](#_bookmark556)

**passing references**

to alter arguments, [191](#_bookmark459)–[194](#_bookmark465) passing by reference, [194](#_bookmark465) passing by value, [193](#_bookmark462)–[194](#_bookmark465) Swap program, [191](#_bookmark459)–[193](#_bookmark462)

constant references, [197](#_bookmark472)–[198](#_bookmark475) Inventory Displayer program,

[195](#_bookmark468)–[196](#_bookmark469)

pitfalls, [196](#_bookmark469)–[197](#_bookmark472)

**PassTime() member function,**

**Critter class,** [**278**](#_bookmark642) **Peek() global function,** [**295**](#_bookmark670) **planning programs,** [**139**](#_bookmark333)**–**[**141**](#_bookmark340)

pseudocode, [139](#_bookmark333)–[140](#_bookmark336)

stepwise reﬁnement, [140](#_bookmark337)–[141](#_bookmark341)

**Play Again 2.0 program,** [**56**](#_bookmark155)**–**[**57**](#_bookmark157) **Play Again program,** [**54**](#_bookmark150)**–**[**55**](#_bookmark152)

**Play() member function,**

**Critter class,** [**280**](#_bookmark644)

**Player class**

BlackJack game, [357,](#_bookmark774) [359,](#_bookmark775)

[368](#_bookmark787)–[369](#_bookmark788)

Game Lobby program, [316](#_bookmark701)–[318](#_bookmark703)

**pointers,** [**223**](#_bookmark524)**–**[**250**](#_bookmark584)

arrays and, [244](#_bookmark571)–[248](#_bookmark578) Array Passer program,

[244](#_bookmark571)–[246](#_bookmark574)

constant pointer, [246](#_bookmark574)–[247](#_bookmark576) passing and returning,

[247](#_bookmark576)–[248](#_bookmark578)

assigning addresses to, [227](#_bookmark533)–[228](#_bookmark536)

constants and, [231](#_bookmark545)–[234](#_bookmark550)

dangling, [301,](#_bookmark682) [325](#_bookmark715)

declaring, [226](#_bookmark530)–[227](#_bookmark533) declaring data member, [307](#_bookmark688) deﬁned, [248](#_bookmark579)

dereferencing, [228](#_bookmark536)–[229](#_bookmark539)

initializing, [227](#_bookmark533)

passing, [234](#_bookmark550)–[238](#_bookmark557)

constant pointer, [237](#_bookmark556)–[238](#_bookmark557) Swap Pointer Version

program, [234](#_bookmark550)–[236](#_bookmark553)

by value, [236](#_bookmark553)–[237](#_bookmark556)

Pointing program, [224](#_bookmark527)–[226](#_bookmark530)

reassigning, [229](#_bookmark539)

returning, [238](#_bookmark557)–[244](#_bookmark571)

Tic-Tac-Toe 2.0 game, [248](#_bookmark578) using with objects, [230](#_bookmark542)–[231](#_bookmark545)

**Pointing program,** [**224**](#_bookmark527)**–**[**226**](#_bookmark530) **Polymorphic Bad Guy program,**

[**347**](#_bookmark758)**–**[**350**](#_bookmark761)

**polymorphism,** [**347**](#_bookmark758)**–**[**380**](#_bookmark802)

abstract classes, [352](#_bookmark764)–[356](#_bookmark772) Abstract Creature program,

[352](#_bookmark764)–[354](#_bookmark767)

declaring pure virtual functions, [354](#_bookmark767)–[355](#_bookmark770)

deriving classes from, [355](#_bookmark770)–[356](#_bookmark772)

Blackjack game, [356](#_bookmark772)–[379](#_bookmark800)

Card class, [361](#_bookmark778)–[363](#_bookmark780)

Deck class, [370](#_bookmark790)–[373](#_bookmark793)

designing classes, [356](#_bookmark772)–[360](#_bookmark776)

Game class, [373](#_bookmark793)–[376](#_bookmark795)

GenericPlayer class, [366](#_bookmark783)–[367](#_bookmark785)

Hand class, [363](#_bookmark780)–[366](#_bookmark783)

House class, [369](#_bookmark788)–[370](#_bookmark790)

main() function, [376](#_bookmark795)–[377](#_bookmark797) overloading the operator()

function, [377](#_bookmark797)–[379](#_bookmark800)

Index 407

planning game logic, [360](#_bookmark776)–[361](#_bookmark778)

Player class, [368](#_bookmark787)–[369](#_bookmark788) deﬁning virtual destructors,

[351](#_bookmark762)–[352](#_bookmark764)

Polymorphic Bad Guy program, [347](#_bookmark758)–[350](#_bookmark761)

using base class pointers to derived class objects, [350](#_bookmark761)–[351](#_bookmark762)

**pop\_back() member function**

adding or removing elements, [138](#_bookmark330)

vectors, [122](#_bookmark296)

**postﬁx increment operator,** [**27**](#_bookmark89)**–**[**28**](#_bookmark90) **precedence level, operators,** [**15**](#_bookmark57) **predicate function,** [**113**](#_bookmark274)

**preﬁx increment operator,** [**27**](#_bookmark89)**–**[**28**](#_bookmark90) **premature optimization,** [**186**](#_bookmark447)

**preprocessor,** [**7**](#_bookmark30)

**priority queue,** [**148**](#_bookmark358)

**priority\_queue STL container,** [**139**](#_bookmark333)

**Private Critter program,** [**264**](#_bookmark613)**–**[**266**](#_bookmark614)

**private keyword,** [**267**](#_bookmark616)

**private members,** [**282**](#_bookmark648)class access levels, [267](#_bookmark616) deﬁned, [340](#_bookmark742)

**procedural programming,** [**283**](#_bookmark651)

**proﬁling code,** [**180**](#_bookmark431)

**program, deﬁned,** [**35**](#_bookmark106)

**protected members,** [**340,**](#_bookmark741) [**379**](#_bookmark800) **prototypes, function**

overview, [153](#_bookmark368)

parameter names and, [159](#_bookmark381) **pScore pointer,** [**224**](#_bookmark527) **pseudocode**

Critter Caretaker game, [276](#_bookmark636) deﬁned, [146](#_bookmark355)

Tic-Tac-Toe game, [204](#_bookmark490) when to use, [148](#_bookmark358)

**pseudorandom number,** [**70,**](#_bookmark185) [**79**](#_bookmark204)

**ptrToElement() function,** [**241**](#_bookmark563)**–**[**242**](#_bookmark566)

**public derivation,** [**379**](#_bookmark800)

**public inheritance,** [**356**](#_bookmark772)

**public keyword,** [**258,**](#_bookmark595) [**266**](#_bookmark614)

**public members,** [**282**](#_bookmark648)class access levels, [266](#_bookmark614) deﬁned, [339](#_bookmark738)

**pure virtual function,** [**380**](#_bookmark802)

**push\_back() member function,** [**292**](#_bookmark665)

adding or removing elements, [138](#_bookmark330)

vectors, [120](#_bookmark289)

**pushing, BlackJack,** [**356**](#_bookmark772)

**puzzle games,** [**107**](#_bookmark260)

##### Q

**queue,** [**148**](#_bookmark358)

**queue STL container,** [**139**](#_bookmark333)

##### R

**rand()function,** [**69**](#_bookmark183)**–**[**70**](#_bookmark185) **random numbers, generating,**

[**68**](#_bookmark181)**–**[**72**](#_bookmark190)

calculating number within range, [71](#_bookmark187)–[72](#_bookmark190)

Die Roller program, [68](#_bookmark181)–[69](#_bookmark183)

rand()function, [69](#_bookmark183)–[70](#_bookmark185)

seeding, [70](#_bookmark185)–[71](#_bookmark187)

**random\_shufﬂe() algorithm,** [**134**](#_bookmark320)**–**[**135,**](#_bookmark323) [**143**](#_bookmark346)

**random-element access,** [**147**](#_bookmark357)

**rank m\_Rank member, BlackJack**

**Card class,** [**359**](#_bookmark775)

**reallocation, vector memory,** [**136**](#_bookmark326) **reassigning pointers,** [**229**](#_bookmark539)

**redundant parentheses,** [**68**](#_bookmark181) **reference operator (&),**

[**189,**](#_bookmark454) [**217**](#_bookmark514)

**references,** [**187**](#_bookmark450)**–**[**217**](#_bookmark514)

versus constant pointers, [232](#_bookmark547) creating, [189](#_bookmark454)–[190](#_bookmark456)

deciding how to pass arguments, [198](#_bookmark475)

deﬁned, [217](#_bookmark515) passing

to alter arguments, [191](#_bookmark459)–[194](#_bookmark465) constant references,

[197](#_bookmark472)–[198](#_bookmark475)

Inventory Displayer pro- gram, [195](#_bookmark468)–[196](#_bookmark469)

pitfalls, [196](#_bookmark469)–[197](#_bookmark472)

versus pointers, [249](#_bookmark581)

Referencing program, [187](#_bookmark450)–[189](#_bookmark454)

returning, [198](#_bookmark475)–[203](#_bookmark488)

altering object through, [202](#_bookmark485) assigning to reference, [202](#_bookmark485) assigning to variable, [202](#_bookmark485) displaying value of, [201](#_bookmark481) Inventory Referencer

program, [199](#_bookmark478)–[200](#_bookmark479)

overview, [200](#_bookmark479)–[201](#_bookmark481)

Tic-Tac-Toe game, [203](#_bookmark488)–[217](#_bookmark514) announceWinner() function, [217](#_bookmark515)

askNumber()function, [209](#_bookmark500)

askYesNo()function, [208](#_bookmark497)–[209](#_bookmark500)

computerMove()function, [213](#_bookmark510)–[216](#_bookmark512)

creating list of functions, [205](#_bookmark491)

displayBoard()function, [210](#_bookmark503)–[211](#_bookmark506)

humanMove()function, [213](#_bookmark510)

humanPiece()function, [209](#_bookmark500)–[210](#_bookmark503)

instructions()function, [208](#_bookmark497)

isLegal()function, [212](#_bookmark507)–[213](#_bookmark510)

main()function, [207](#_bookmark495)–[208](#_bookmark497)

opponent()function, [210](#_bookmark503)

pseudocode, [204](#_bookmark490)

representing data, [204](#_bookmark490)–[205](#_bookmark491)

setting up, [205](#_bookmark491)–[207](#_bookmark495)

winner()function, [211](#_bookmark506)–[212](#_bookmark507) values

accessing, [190](#_bookmark456)

altering, [190](#_bookmark456)–[191](#_bookmark459)

**Referencing program,** [**187**](#_bookmark450)**–**[**189**](#_bookmark454)

**refToElement() function header,** [**200**](#_bookmark479)**–**[**201,**](#_bookmark481) [**240**](#_bookmark561)

**relational operators,** [**40*tbl***](#_bookmark119)**,** [**44**](#_bookmark129)

**reserve()member function,** [**137**](#_bookmark328)

**restricting pointers,** [**243,**](#_bookmark568) [**251**](#_bookmark585)

**return statement,** [**50,**](#_bookmark141) [**158,**](#_bookmark379)

[**184**](#_bookmark441)**–**[**185**](#_bookmark444)

**return type, constructor,** [**262**](#_bookmark607) **return values, function,** [**157**](#_bookmark377)**–**[**158**](#_bookmark379)

return statement, [158](#_bookmark379) specifying return type, [157](#_bookmark377) using, [158](#_bookmark379)

**returning pointers,** [**238**](#_bookmark557)**–**[**244**](#_bookmark571)

altering object through, [243](#_bookmark568)–[244](#_bookmark571)

assigning to pointer, [242](#_bookmark566) assigning to variable value

pointed to by returned pointer, [242](#_bookmark566)–[243](#_bookmark568)

Inventory Pointer program, [239](#_bookmark560)–[240](#_bookmark561)

overview, [240](#_bookmark561)–[241](#_bookmark563)

using to display value, [241](#_bookmark563)–[242](#_bookmark566)

**returning references,** [**198**](#_bookmark475)**–**[**203,**](#_bookmark488) [**220**](#_bookmark520)altering object through, [202](#_bookmark485) assigning, [202](#_bookmark485)

assigning to reference, [202](#_bookmark485) assigning to variable, [202](#_bookmark485) displaying value of, [201](#_bookmark481) Inventory Referencer program,

[199](#_bookmark478)–[200](#_bookmark479)

overview, [200](#_bookmark479)–[201](#_bookmark481)

408 Index

**reusability, class,** [**332**](#_bookmark725)**–**[**333**](#_bookmark727) **role-playing game (RPG),** [**72**](#_bookmark190) **run-time errors,** [**4**](#_bookmark23)

##### S

**scope resolution operator (::), 9,** [**125,**](#_bookmark302) [**259,**](#_bookmark599) [**345**](#_bookmark753)

**scopes,** [**161**](#_bookmark385)**–**[**166**](#_bookmark395)

hiding global variables, [169](#_bookmark402) nested, [165](#_bookmark393)–[166,](#_bookmark396) [185](#_bookmark445)

Scoping program, [161](#_bookmark385)–[163](#_bookmark388)

separate, [163](#_bookmark388)–[164](#_bookmark390)

**Score Rater program,** [**41**](#_bookmark123)**–**[**42**](#_bookmark124)

version 2.0, [46](#_bookmark134)–[47](#_bookmark135)

version 3.0, [49](#_bookmark140)–[50](#_bookmark141)

**seeding random number generator,** [**70**](#_bookmark185)**–**[**71,**](#_bookmark187) [**79,**](#_bookmark204) [**134**](#_bookmark320)

**self-documenting code,** [**21**](#_bookmark74) **semicolon (;)**

if statement, [43](#_bookmark126)

terminating statements, [9](#_bookmark38)

**separate scopes,** [**163**](#_bookmark388)**–**[**164**](#_bookmark390)

**sequential container,** [**138,**](#_bookmark330) [**147**](#_bookmark357) **set STL container,** [**139**](#_bookmark333) **shallow copy,** [**310**](#_bookmark695)

**short int type,** [**19*tbl***](#_bookmark68)

**short modiﬁer,** [**18**](#_bookmark65)

**signed modiﬁer,** [**18**](#_bookmark65) **Simple Boss 2.0 program,**

[**338**](#_bookmark736)**–**[**339**](#_bookmark737)

**Simple Boss program,** [**333**](#_bookmark727)**–**[**335**](#_bookmark728) **Simple Critter program,** [**256**](#_bookmark592)**–**[**257**](#_bookmark593) **64-element array,** [**103**](#_bookmark253)

**size()member function,** [**92,**](#_bookmark227) [**120**](#_bookmark289)

**skYesNo2() function,** [**158**](#_bookmark379)

**slicing, OOP,** [**381**](#_bookmark804)

**software reuse,** [**161**](#_bookmark385)

**sort() algorithm,** [**135**](#_bookmark323)

**source code,** [**2**](#_bookmark19)

**srand() function,** [**71,**](#_bookmark187) [**78**](#_bookmark203)

**stack,** [**147,**](#_bookmark357) [**296,**](#_bookmark673) [**325**](#_bookmark715)

**stack STL container,** [**139**](#_bookmark333) **standard I/O,** [**5**](#_bookmark25)

displaying text, [8](#_bookmark35)–[9](#_bookmark38)

std namespace, [10](#_bookmark41)–[13](#_bookmark52) Game Over 2.0 program,

[10](#_bookmark41)–[11](#_bookmark45)

Game Over 3.0 program, [11](#_bookmark45)–[12](#_bookmark48)

using declarations, [12](#_bookmark48)–[13](#_bookmark52)

using directive, [11,](#_bookmark45) [13](#_bookmark52)

**standard library, 5,** [**35**](#_bookmark106) **Standard Template Library.**

***See* STL**

**Static Critter program,** [**270**](#_bookmark623)**–**[**271**](#_bookmark624)

**static data members,** [**282**](#_bookmark648)accessing, [272](#_bookmark626)–[273](#_bookmark629) constant member function

and, [269](#_bookmark620)

declaring, [272](#_bookmark626)

initializing, [272](#_bookmark626)

Static Critter program, [270](#_bookmark623)–[271](#_bookmark624)

**static keyword**

declaring static data members, [272](#_bookmark626)

declaring static member function, [273](#_bookmark629)

**static member functions,** [**282**](#_bookmark648)

calling, [273](#_bookmark629)–[274](#_bookmark633)

declaring, [273](#_bookmark629)

deﬁning, [273](#_bookmark630)

**std namespace,** [**10**](#_bookmark41)**–**[**13**](#_bookmark52)

algorithms and, [133](#_bookmark317)

deﬁned, [36](#_bookmark110)

Game Over 2.0 program, [10](#_bookmark41)–[11](#_bookmark45)

Game Over 3.0 program, [11](#_bookmark45)–[12](#_bookmark48)

using declarations, [12](#_bookmark48)–[13](#_bookmark52)

using directive, [11,](#_bookmark45) [13](#_bookmark52)

**stepwise reﬁnement** deﬁned, [146](#_bookmark355) when to use, [148](#_bookmark358)

**STL (Standard Template Library),** [**115**](#_bookmark278)**–**[**149**](#_bookmark361)

algorithms, [131](#_bookmark315)–[135](#_bookmark323)

ﬁnd(), [134](#_bookmark320)

High Scores program, [131](#_bookmark315)–[133](#_bookmark317)

preparing to use, [133](#_bookmark317)

random\_shufﬂe(), [134](#_bookmark320)–[135](#_bookmark323)

sort(), [135](#_bookmark323)

containers, [138](#_bookmark330)–[139](#_bookmark333)

Hangman game, [141](#_bookmark340)–[145](#_bookmark352)

ending, [145](#_bookmark352) entering main loop,

[143](#_bookmark346)–[144](#_bookmark349)

getting player's guess, [144](#_bookmark349)–[145](#_bookmark352)

initializing variables and constants, [143](#_bookmark346)

planning, [141](#_bookmark340)–[142](#_bookmark343)

setting up, [142](#_bookmark343)

iterators, [123](#_bookmark301)–[131](#_bookmark315) accessing member

functions of vector element, [129](#_bookmark310)–[130](#_bookmark312) changing value of vector element, [128](#_bookmark307)–[129](#_bookmark310)

declaring, [125](#_bookmark302)–[126](#_bookmark304)

erase()vector member function, [130](#_bookmark312)–[131](#_bookmark315)

Hero's Inventory 3.0

program, [123](#_bookmark301)–[125](#_bookmark302)

insert()vector member function, [130](#_bookmark312)

looping through vector, [126](#_bookmark304)–[128](#_bookmark307)

overview, [115](#_bookmark278)–[116](#_bookmark282)

planning programs, [139](#_bookmark333)–[141](#_bookmark340)

pseudocode, [139](#_bookmark333)–[140](#_bookmark336) stepwise reﬁnement,

[140](#_bookmark336)–[141](#_bookmark340)

vectors, [116](#_bookmark281)–[122,](#_bookmark297) [136](#_bookmark327)–[138](#_bookmark331) calling member functions of

element, [121](#_bookmark292)–[122](#_bookmark296)

clear()member function, [122](#_bookmark296)

**STL (Standard Template Library) (*continued* )**

declaring, [119](#_bookmark286)–[120](#_bookmark289) element insertion and

deletion, [138](#_bookmark330)

empty()member function, [122](#_bookmark296)

growth, [136](#_bookmark326)–[137](#_bookmark328)

Hero's Inventory 2.0

program, [117](#_bookmark284)–[119](#_bookmark286)

indexing, [121](#_bookmark292)

pop\_back()member function, [122](#_bookmark296)

preparing to use, [119](#_bookmark286)

push\_back() member function, [120](#_bookmark289)

size()member function, [120](#_bookmark289)

**strategy games,** [**216**](#_bookmark512)

**streams,** [**34,**](#_bookmark104) [**182**](#_bookmark436)

**string argument,** [**175**](#_bookmark419)

**string literal,** [**8**](#_bookmark35)

**string m\_Name member, BlackJack GenericPlayer class,** [**359**](#_bookmark775)

**string objects,** [**33,**](#_bookmark102) [**38,**](#_bookmark114) [**89**](#_bookmark222)**–**[**96,**](#_bookmark238) [**135**](#_bookmark323)

array, [96](#_bookmark238)–[97](#_bookmark240)

concatenating, [92](#_bookmark227)

creating, [91](#_bookmark224)

deﬁned, [110](#_bookmark269)

empty()member function, [96](#_bookmark238)

erase()member function, [95](#_bookmark235)

ﬁnd() member function, [94](#_bookmark233)–[95](#_bookmark235)

indexing, [93](#_bookmark230)

iterating through, [93](#_bookmark230)–[94](#_bookmark233)

size()member function, [92](#_bookmark227) String Tester program, [89](#_bookmark222)–[91](#_bookmark224)

**String Tester program,** [**89**](#_bookmark222)**–**[**91**](#_bookmark224)

Index 409

**strings**

C-style strings, [101](#_bookmark248)–[102](#_bookmark250)

overview, [8](#_bookmark35)

**Stroustrup, Bjarne,** [**1**](#_bookmark16)

**structure, deﬁned,** [**284**](#_bookmark653) **subclass. *See* derived classes**

**subscripting operator ([ ]),** [**93,**](#_bookmark230) [**99,**](#_bookmark243)

[**121,**](#_bookmark292) [**147**](#_bookmark357)

**subtraction (**-**) operator,** [**14**](#_bookmark54)

**suit m\_Suit member, BlackJack**

**Card class,** [**359**](#_bookmark775)

**superclass,** [**335.**](#_bookmark728) ***See also* base class Swap Pointer Version program,**

[**234**](#_bookmark550)**–**[**236**](#_bookmark553)

**Swap program,** [**191**](#_bookmark459)**–**[**193**](#_bookmark462)

**switch statements,** [**39,**](#_bookmark117) [**51**](#_bookmark144)**–**[**54,**](#_bookmark150) [**77**](#_bookmark201)creating ways to branch, [54](#_bookmark150) deﬁning variables inside, [166](#_bookmark395) Menu Chooser program, [52](#_bookmark146)–[53](#_bookmark147)

**syntactic sugar,** [**130,**](#_bookmark312) [**250**](#_bookmark584)

**syntax errors,** [**4**](#_bookmark23)

##### T

**tab (\t),** [**23**](#_bookmark79)

**Taking Damage program,** [**177**](#_bookmark424)**–**[**179**](#_bookmark428)

**Talk() member function,**

**Critter class,** [**279**](#_bookmark643)

**tellStory()function,** [**183**](#_bookmark439)

**terminating statements,** [**9**](#_bookmark38)

**test expression, for loop,** [**82**](#_bookmark211) **text, displaying through standard**

**output,** [**8**](#_bookmark35)**–**[**9**](#_bookmark38)

**this pointer,** [**326**](#_bookmark718)

**Tic-Tac-Toe 2.0 game,** [**248**](#_bookmark578) **Tic-Tac-Toe Board program,**

[**103**](#_bookmark253)**–**[**105**](#_bookmark255)

**Tic-Tac-Toe game,** [**203**](#_bookmark488)**–**[**217**](#_bookmark514)

announceWinner()function, [217](#_bookmark515)

askNumber()function, [209](#_bookmark500)

askYesNo()function, [208](#_bookmark497)–[209](#_bookmark500)

computerMove()function, [213](#_bookmark510)–[216](#_bookmark512)

creating list of functions, [205](#_bookmark491)

displayBoard()function, [210](#_bookmark503)–[211](#_bookmark506)

humanMove()function, [213](#_bookmark510)

humanPiece()function, [209](#_bookmark500)–[210](#_bookmark503)

instructions()function, [208](#_bookmark497)

isLegal()function, [212](#_bookmark507)–[213](#_bookmark510)

main()function, [207](#_bookmark495)–[208](#_bookmark497)

opponent()function, [210](#_bookmark503)

pseudocode, [204](#_bookmark490)

representing data, [204](#_bookmark490)–[205](#_bookmark491)

setting up, [205](#_bookmark491)–[207](#_bookmark495)

winner()function, [211](#_bookmark506)–[212](#_bookmark507)

**toupper() function,** [**155**](#_bookmark374)

**triple() function,** [**176**](#_bookmark421)

**Triple program,** [**174**](#_bookmark415)**–**[**176**](#_bookmark421)

**true values**

interpreting, [43](#_bookmark126)

testing, [42](#_bookmark124)–[43](#_bookmark126)

**truth,** [**39**](#_bookmark117)**–**[**40**](#_bookmark120)

**two-dimensional array,** [**103**](#_bookmark253)

**type casting,** [**372**](#_bookmark792) **types**

*See also* classes

deﬁning new names, [23](#_bookmark79) fundamental, [18](#_bookmark65)

modiﬁers, [18](#_bookmark66)–[19](#_bookmark68)

understanding which to use, [24](#_bookmark83)

##### U

**unsigned int type,** [**19*tbl***](#_bookmark68)**unsigned long int type,** [**19*tbl***](#_bookmark68)**unsigned modiﬁer,** [**18**](#_bookmark65) **unsigned short int type,** [**19*tbl***](#_bookmark68)**uppercase() function,** [**145**](#_bookmark352)

**using declarations,** [**12**](#_bookmark48)**–**[**13**](#_bookmark52)

**using directive,** [**11,**](#_bookmark45) [**13**](#_bookmark52)

##### V

**values**

*See also* assigning values data member, accessing, [260](#_bookmark603) displaying, [22](#_bookmark76)–[23](#_bookmark79)

references

accessing, [190](#_bookmark456)

altering, [190](#_bookmark456)–[191](#_bookmark459) returning from main()

function, [10](#_bookmark41)

**variables,** [**16**](#_bookmark61)**–**[**24**](#_bookmark83)

assigning returning references, [202](#_bookmark485)

assigning values, [21](#_bookmark74)–[22](#_bookmark76)

to Boolean variables, [22](#_bookmark76)

to character variables, [21](#_bookmark74)–[22](#_bookmark76) to ﬂoating point variables, [21](#_bookmark74) to integer variables, [21](#_bookmark74)

declaring, [19](#_bookmark69)–[20](#_bookmark71) deﬁning new names

for types, [23](#_bookmark79)

displaying values, [22](#_bookmark76)–[23](#_bookmark79)

fundamental types, [18](#_bookmark65) Game Stats program, [16](#_bookmark61)–[17](#_bookmark62) getting user input, [23](#_bookmark79) global. *See* global variables initializing, [22](#_bookmark76)

naming, [20](#_bookmark71)–[21](#_bookmark74) performing arithmetic

operations, [24](#_bookmark83)–[29](#_bookmark93)

altering value of variable, [26](#_bookmark85) combined assignment

operators, [26](#_bookmark85)–[27](#_bookmark89)

decrement operator, [27](#_bookmark89)–[28](#_bookmark90) Game Stats 2.0 program,

[24](#_bookmark83)–[26](#_bookmark85)

increment operator, [27](#_bookmark89)–[28](#_bookmark90) integer wrap around, [28](#_bookmark90)–[29](#_bookmark93)

scopes, [161](#_bookmark385)

type modiﬁers, [18](#_bookmark66)–[19](#_bookmark68) understanding which types to

use, [24](#_bookmark83)

**vector member functions,** [**115**](#_bookmark278)

**vector objects,** [**115**](#_bookmark278) **vector STL container,** [**139**](#_bookmark333) **vector***<***Card\*> m\_Cards**

**member, BlackJack Hand class,** [**359**](#_bookmark775)

**vector***<***Player> m\_Players member, BlackJack Game class,** [**360**](#_bookmark776)

**vectors,** [**116**](#_bookmark282)**–**[**122,**](#_bookmark296) [**136**](#_bookmark326)**–**[**138**](#_bookmark330)

versus arrays, [146](#_bookmark354)–[147](#_bookmark357) calling member functions of

element, [121](#_bookmark292)–[122](#_bookmark296)

clear()member function, [122](#_bookmark296)

declaring, [119](#_bookmark286)–[120](#_bookmark289)

deﬁned, [146](#_bookmark355)

element insertion and deletion, [138](#_bookmark330)

empty()member function, [122](#_bookmark296)

growth, [136](#_bookmark326)–[137](#_bookmark328)

capacity()member function, [136](#_bookmark326)–[137](#_bookmark328)

reserve()member function, [137](#_bookmark328)

Hero's Inventory 2.0 program, [117](#_bookmark284)–[119](#_bookmark286)

indexing, [121](#_bookmark292)

looping iterators through, [126](#_bookmark304)–[128](#_bookmark307)

altering iterator, [128](#_bookmark307) calling begin()vector

member function, [127](#_bookmark306) calling end()vector member

function, [127](#_bookmark306)–[128](#_bookmark307)

dereferencing iterator, [128](#_bookmark307)

pop\_back()member function, [122](#_bookmark296)

preparing to use, [119](#_bookmark286)

push\_back() member function, [120](#_bookmark289)

size()member function, [120](#_bookmark289)

410 Index

**virtual base class member functions,** [**344**](#_bookmark750)**–**[**345**](#_bookmark753)

**virtual bool IsHitting() const = 0 member, BlackJack GenericPlayer class,** [**359**](#_bookmark775)

**virtual bool IsHitting() const member**

BlackJack House class, [360](#_bookmark776) BlackJack Player class, [359](#_bookmark775)

**virtual destructors,** [**347**](#_bookmark758)

**virtual functions,** [**354**](#_bookmark767)**–**[**355,**](#_bookmark770) [**380**](#_bookmark802)

**virtual keyword,** [**268,**](#_bookmark618) [**365**](#_bookmark782)

**void Add(Card\* pCard) member,**

**BlackJack Hand class,** [**359**](#_bookmark775)

**void AdditionalCards member,**

**BlackJack Deck class,** [**360**](#_bookmark776)

**void Bust() const member, BlackJack GenericPlayer class,** [**359**](#_bookmark775)

**void Clear() member, BlackJack**

**Hand class,** [**359**](#_bookmark775)

**void Deal(Hand& aHand) member, BlackJack Deck class,** [**360**](#_bookmark776)

**void Flip() member, BlackJack**

**Card class,** [**359**](#_bookmark775)

**void FlipFirstCard() member,**

**BlackJack House class,** [**360**](#_bookmark776)

**void Lose() const member,**

**BlackJack Player class,** [**359**](#_bookmark775)

**void Play() member, BlackJack**

**Game class,** [**360**](#_bookmark776)

**void Populate() member,**

**BlackJack Deck class,** [**360**](#_bookmark776)

**void Push() const member,**

**BlackJack Player class,** [**359**](#_bookmark775)

**void setDisplay prototype,** [**173**](#_bookmark412)**–**[**174**](#_bookmark415)

**void Shufﬂe() member,**

**BlackJack Deck class,** [**360**](#_bookmark776)

**void Win() const member, BlackJack Player class,** [**359**](#_bookmark775)

##### W

**while (true) loop,** [**60,**](#_bookmark164) [**79**](#_bookmark204)

**while loops,** [**54**](#_bookmark150)**–**[**56,**](#_bookmark155) [**77**](#_bookmark201)

deﬁning variables inside, [166](#_bookmark395)

versus for loop, [111](#_bookmark271) overview, [55](#_bookmark152)–[56](#_bookmark155)

Play Again program, [54](#_bookmark150)–[55](#_bookmark152)

**whitespace,** [**7,**](#_bookmark30) [**37,**](#_bookmark112) [**226**](#_bookmark530) **Win32 Application**

**Wizard – Application Settings,** [**385**](#_bookmark811)**ﬁ**[**g**](#_bookmark811)

**winner()function, Tic-Tac-Toe game,** [**211**](#_bookmark506)**–**[**212**](#_bookmark507)

**Word Jumble game,** [**106**](#_bookmark258)**–**[**107**](#_bookmark260)

choosing word to jumble, [107](#_bookmark260)–[108](#_bookmark262)

entering game loop, [109](#_bookmark265)–[110](#_bookmark268) exiting, [110](#_bookmark268)

jumbling word, [108](#_bookmark262)–[109](#_bookmark265)

setting up, [107](#_bookmark260)

welcoming player, [109](#_bookmark265)

##### Y

**Yes or No program,** [**155**](#_bookmark374)**–**[**157**](#_bookmark377)